(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 821 211 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
22.08.2007 Bulletin 2007/34

(51) Int Cl.:
G06F 9/48 *(2006.01)* ***G06F 15/80*** *(2006.01)*
G06F 9/32 *(2006.01)*

(21) Application number: **07250647.0**

(22) Date of filing: **15.02.2007**

(84) Designated Contracting States:
AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HU IE IS IT LI LT LU LV MC NL PL PT RO SE SI SK TR
Designated Extension States:
AL BA HR MK YU

(30) Priority: **16.02.2006 US 355495**
16.02.2006 US 355513
31.03.2006 US 788265 P
03.05.2006 US 797345 P
26.05.2006 US 441818
30.06.2006 US 818084 P
29.09.2006 US 849498 P
12.01.2007 US 653187
26.05.2006 US 441784
26.05.2006 US 441812

(71) Applicant: **Technology Properties Limited**
Cupertino CA 95014 (US)

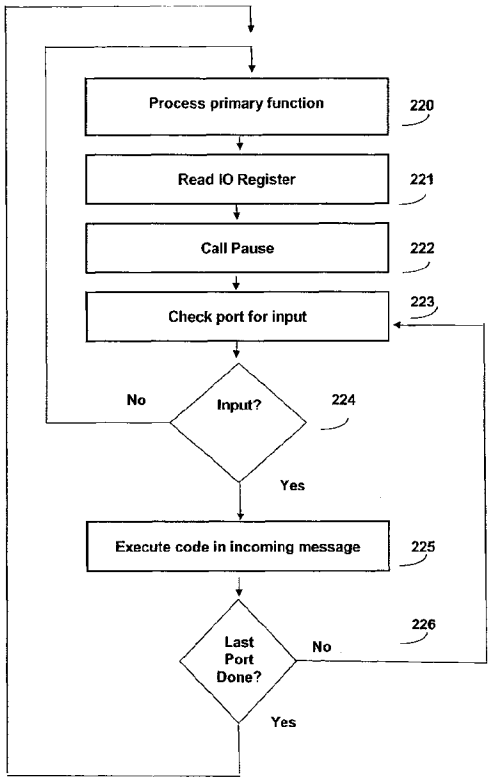
(72) Inventors:
- **Moore, Charles H.**
California 96125 (US)
- **Fox, Jeffrey Arthur**
California 94710 (US)
- **Rible, John W.**
Santa Cruz, California 95060-4215 (US)

(74) Representative: **Beresford, Keith Denis Lewis et al**
Beresford & Co
16 High Holborn
London WC1V 6BX (GB)

(54) Cooperative multitasking method in a multiprocessor system

(57) A microprocessor system in which an array of processors communicates more efficiently through the use of a worker mode function. Processors that are not currently executing code remain in an inactive but alert state until a task is sent to them by an adjacent processor. Processors can also be programmed to temporarily suspend a task to check for incoming tasks or messages.

Fig. 12



EP 1 821 211 A2

Description

[0001] This application claims priority to provisional application number 60/849,498, entitled "SEAForth Applications Guide," filed Sept. 29, 2006, and also claims priority to provisional application number 60/818,084, filed June 30, 2006; and is a continuation-in-part of the application entitled, "Method and Apparatus for Monitoring Inputs to a Computer," serial number 11/441,818, filed May 26, 2006; and claims priority to provisional application number 60/797,345, filed May 3, 2006, and also claims priority to provisional application number 60/788,265, filed March 31, 2006; and is a continuation-in-part of the application entitled, "Asynchronous Power Saving Computer," serial number 11/355,513, filed Feb. 16, 2006. All of the cited applications above are incorporated herein by reference in their entireties.

BACKGROUND OF THE INVENTION

Field of the Invention

[0002] The present invention relates to the field of computers and computer processors, and more particularly to a method and means for allowing a computer to execute instructions as they are received from an external source without first storing said instructions, and an associated method to facilitate communications between computers and the ability of a computer to use the available resources of another computer. The predominant current usage of the present inventive direct execution method and apparatus is in the combination of multiple computers on a single microchip, wherein operating efficiency is important not only because of the desire for increased operating speed but also because of the power savings and heat reduction that are a consequence of the greater efficiency.

Description of the Background Art

[0003] In the art of computing, processing speed is a much desired quality, and the quest to create faster computers and processors is ongoing. However, it is generally acknowledged in the industry that the limits for increasing the speed in microprocessors are rapidly being approached, at least using presently known technology. Therefore, there is an increasing interest in the use of multiple processors to increase overall computer speed by sharing computer tasks among the processors.

[0004] The use of multiple processors creates a need for communication between the processors. Therefore, there is a significant portion of time spent in transferring instructions and data between processors. Each additional instruction that must be executed in order to accomplish this places an incremental delay in the process which, cumulatively, can be very significant. The conventional method for communicating instructions or data from one computer to another involves first storing the data or instruction in the receiving computer and then, subsequently calling it for execution (in the case of an instruction) or for operation thereon (in the case of data).

[0005] In the prior art it is known that it is necessary to "get the attention" of a computer from time to time. Even though a computer may be busy with one task, another time-sensitive task requirement can occur that may necessitate temporarily diverting the computer away from the first task. Examples include, but are not limited to, providing input to a computer. In such cases, the computer might need to temporarily acknowledge the input and/or react in accordance with the input. Then, the computer will either continue what it was doing before the input or else change what it was doing based upon the input. Although an external input is used as an example here, the same situation occurs when there is a potential conflict for the attention of the ALU between internal aspects of the computer, as well.

[0006] When receiving data and changes in status from I/O ports, there have been two methods available in the prior art. One has been to "poll" the port, which involves reading the status of the port at fixed intervals to determine whether any data has been received or a change of status has occurred. However, polling the port consumes considerable time and resources. A better alternative has often been the use of "interrupts". When using interrupts, a processor can go about performing its assigned task and then, when an I/O port/device needs attention or the status changes, it sends an Interrupt Request (IRQ) to the processor. Once the processor receives an Interrupt Request, it will, for example, finish its current instruction, place a few things on the stack, and then execute the appropriate Interrupt Service Routine (ISR). Once the ISR has finished, the processor returns to where it left off. When using this method, the processor doesn't have to waste time, looking to see if the I/O device is in need of attention, but rather the device will only service the interrupt when it needs attention. However, the use of interrupts is far less than desirable in many cases, since there can be a great deal of overhead associated with the use of interrupts. For example, each time an interrupt occurs, a computer may have to temporarily store certain data relating to the task it was previously trying to accomplish, then load data pertaining to the interrupt, and then reload the data necessary for the prior task once the interrupt is handled.

[0007] An improvement to the above systems can be found using a system based on the Forth computer language. Forth systems have been able to have more than one "thread" of code executing at one time. This is often called a cooperative round-robin. The order in which the threads get a turn using the central processing unit (CPU) is fixed; for example, thread 4 always gets its turn after thread 3 and before thread 5. Each thread is allowed to keep the CPU as long as it wants to, and then relinquish it voluntarily. The thread does this by calling the word PAUSE. Only a few data items need to be saved during a PAUSE function in order for the original task to be re-

stored, as opposed to large contexts that need to be saved during an interrupt function.

[0008] Each thread may or may not have work to do. If task 4 has work to do and the task before it in the round-robin (task 3) calls PAUSE, then task 4 will wake up and work until it decides to PAUSE again. If task 4 has no work to do, it passes control on to task 5. When a task calls a word which will perform an input/output function, and will therefore need to wait for the input/output to finish, a PAUSE is built into the input/output call.

[0009] The predictability of PAUSE allows for very efficient code. Frequently, a Forth based cooperative round-robin can give every existing thread a turn at the CPU in less time than it would take a pre-emptive multitasker to decide who should get the CPU next. However, a particular task may tend to overwhelm or overtake the CPU.

[0010] Another area of operating efficiency can be found by minimizing leakage current in a computer system. In the quest to make devices smaller, leakage current increases as insulation layers become thinner. In the near future, leakage power could reach as high as 50% of the active power. One attempt to curb leakage current can be found by utilizing sleep transistors. Sleep transistors act like a switch by isolating or disconnecting the power supply and blocks of logic when they are not needed. This can reduce leakage power by a factor of 2-1000 times, as disclosed in an article entitled *How to Provide a Power-Efficient Architecture,* by Bob Crepps, published in Embedded.com, July 24, 2006.

[0011] However, there are several drawbacks in using sleep transistors. Many factors need to be considered in a system in order to incorporate efficient use of these transistors. The threshold voltage of a sleep transistor needs to be large; otherwise, the sleep transistor will have a high leakage current. This requires modification in the CMOS technology process to support both a high threshold voltage device for the sleep transistor and a low threshold voltage device for the logic gates. In addition, a large sleep transistor increases the area overhead and the dynamic power consumed for turning the transistor on and off.

[0012] To guarantee the proper functionality of a circuit, the sleep transistors have to be carefully sized to decrease their voltage drop while they are on. Two gates that switch at different times can share a sleep transistor. However, this is not practical for large circuits. Algorithms are necessary to determine the best implementation of sleep transistors for large circuits.

[0013] Other problems associated with sleep transistors include generating noise in the circuits, and a loss of data when used to disconnect flip flops from ground or supply voltage, as disclosed in an article entitled Standby and Active Leakage Current Control and Minimization in CMOS VLSI Circuits, by Farzan Fallah, et al., published in 2004. A method and/or apparatus is needed to reduce leakage current and provide a more efficient and less problematic computer processor system.

Summary of Invention

[0014] It would be useful to reduce the number of steps required to transmit, receive, and then use information in the form of data or instructions between computers. It would also be desirable to reduce or eliminate the time and resources consumed during an interrupt. In addition, it would be advantageous to expand the PAUSE function beyond one CPU. However, to the inventor's knowledge, no prior art system has streamlined the above described processes in a significant manner.

[0015] A computer processor array is disclosed in which power usage and heat dissipation are minimized, and computing efficiency is maximized. This is realized in part by a computer processor array, in which processors, also called nodes or cores become inactive but alert when not in an operating mode. The inactive node or core consumes essentially no power while inactive, and becomes active when an adjacent node or pin attempts to communicate with it. After execution of an incoming task, the node will go back to an inactive state until another task is sent to the node.

[0016] Array efficiency is also realized when a core is currently executing code or instructions, and a neighboring core communicates to the executing core. Rather then interrupting the executing core as in a conventional computing system, cores can be programmed to occasionally pause to check for incoming messages. If an incoming message awaits, then the executing core can act on the incoming message after pausing, and then continue with its original task.

[0017] These and other objects and advantages of the present invention will become clear to those skilled in the art in view of the description of modes of carrying out the invention, and the industrial applicability thereof, as described herein and as illustrated in the several figures of the drawings. The objects and advantages listed are not an exhaustive list of all possible advantages of the invention. Moreover, it will be possible to practice the invention even where one or more of the intended objects and/or advantages might be absent or not required in the application.

[0018] Further, those skilled in the art will recognize that various embodiments of the present invention may achieve one or more, but not necessarily all, of the described objects and/or advantages. Accordingly, the objects and/or advantages described herein are not essential elements of the present invention, and should not be construed as limitations.

BRIEF DESCRIPTION OF THE DRAWINGS

[0019]

Fig. 1 is a diagrammatic view of a computer processor array, according to the present invention;

Fig. 2 is a detailed diagram showing a subset of the computer processors of Fig. 1 and a more detailed

view of the interconnecting data buses of Fig. 1;
Fig. 3 is a block diagram depicting a general layout of one of the processors of Figs. 1 and 2;
Fig. 4 is a diagrammatic representation of an instruction word according to the present inventive application;
Fig. 5 is a flow diagram depicting an example of a micro-loop according to the present invention;
Fig. 6 is a flow diagram depicting an example of the inventive method for executing instructions from a port;
Fig. 7 is a flow diagram depicting an example of the inventive improved method for alerting a processor;
Fig. 8 is a flow diagram depicting a method for awakening a processor and sending input to an executing processor;
Fig. 9 is a diagrammatic view of the processor array of Fig. 1 with the processor or node identification and sharing ports with adjacent processors utilizing mirroring;
Fig. 9a is a partial view of Fig. 9 with additional port detail;
Fig. 10 is a partial view of an I/O register;
Fig. 11 is a flow diagram depicting a worker mode loop; and
Fig. 12 is a flow diagram depicting an executing processor with a PAUSE routine.

Detailed Description

[0020] This invention is described with reference to the figures, in which like numbers represent the same or similar elements. While this invention is described in terms of modes for achieving this invention's objectives, it will be appreciated by those skilled in the art that variations may be accomplished in view of these teachings without deviating from the spirit or scope of the presently claimed invention.

[0021] The embodiments and variations of the invention described herein, and/or shown in the drawings, are presented by way of example only and are not limiting as to the scope of the invention. Unless otherwise specifically stated, individual aspects and components of the invention may be omitted or modified for a variety of applications while remaining within the spirit and scope of the claimed invention, since it is intended that the present invention is adaptable to many variations.

[0022] The following invention describes processors (also called computers, nodes, or cores) that operate in a mode referred to as "asleep but alert" or "inactive but alert" which both refer to a mode of operation in which the processor's functions have been temporarily suspended, halted, or ceased where essentially no power is being utilized. At the same time, the processor is alert or in a state of readiness to immediately begin processing functions when instructed to do so. When the inactive processor receives processing instructions, it is referred to as "being awakened" or "becoming activated."

[0023] A mode for carrying out the invention is given by an array of individual computer processors. The array is depicted in a diagrammatic view in Fig. 1 and is designated therein by the general reference character 10. The processor array 10 has a plurality (twenty four in the example shown) of computer processors 12 (sometimes also referred to as "cores" or "nodes" in the example of an array). In the example shown, all of the processors 12 are located on a single die 14. According to the present invention, each of the processors 12 is a generally independently functioning processor, as will be discussed in more detail hereinafter. The processors 12 are interconnected by a plurality (the quantities of which will be discussed in more detail hereinafter) of interconnecting data buses 16. In this example, the data buses 16 are bidirectional asynchronous high speed parallel data buses, although it is within the scope of the invention that other interconnecting means might be employed for the purpose. In the present embodiment of the array 10, not only is data communication between the processors 12 asynchronous, but the individual processors 12 also operate in an internally asynchronous mode. This has been found by the inventor to provide important advantages. For example, since a clock signal does not have to be distributed throughout the processor array 10, a great deal of power is saved. Furthermore, not having to distribute a clock signal eliminates many timing problems that could limit the size of the array 10 or cause other known difficulties. Also, the fact that the individual processors operate asynchronously saves a great deal of power, since each processor will use essentially no power when it is not executing instructions because there is no clock running therein.

[0024] One skilled in the art will recognize that there will be additional components on the die 14 that are omitted from the view of Fig. 1 for the sake of clarity. Such additional components include, but are not limited to power buses, external connection pads, and other such common aspects of a microprocessor chip.

[0025] Processor 12e is an example of one of the processors 12 that is not on the periphery of the array 10. That is, processor 12e has four orthogonally adjacent processors 12a, 12b, 12c and 12d, although it is within the scope of this invention that more than four adjacent processors could be utilized. This grouping of processors 12a through 12e will be used, by way of example, hereinafter in relation to a more detailed discussion of the communications between the processors 12 of the array 10. As can be seen in the view of Fig. 1, interior processors such as processor 12e will have at lease four other processors 12 with which they can directly communicate via the buses 16. In the following discussion, the principles discussed will apply to all of the processors 12 except that the processors 12 on the periphery of the array 10 will be in direct communication with only three or, in the case of the corner processors 12, only two other of the processors 12.

[0026] Fig. 2 is a more detailed view of a portion of Fig.

1 showing only some of the processors 12 and, in particular, processors 12a through 12e, inclusive. The view of Fig. 2 also reveals that the data buses 16 each have a read line 18, a write line 20 and a plurality (eighteen, in this example) of data lines 22. The data lines 22 are capable of transferring all the bits of one eighteen-bit instruction word simultaneously in parallel. It should be noted that, in one embodiment of the invention, some of the processors 12 are mirror images of adjacent processors. However, whether the processors 12 are all oriented identically or as mirror images of adjacent processors is not a limiting aspect of this presently described invention.

[0027] According to one aspect of the present inventive method, a processor 12, such as the processor 12e can set high one, two, three or all four of its read lines 18 such that it is prepared to receive data from the respective one, two, three or all four adjacent processors 12. Similarly, it is also possible for a processor 12 to set one, two, three or all four of its write lines 20 high.

[0028] When one of the adjacent processors 12a, 12b, 12c or 12d sets a write line 20 between itself and the processor 12e high, if the processor 12e has already set the corresponding read line 18 high, then a word is transferred from that processor 12a, 12b, 12c or 12d to the processor 12e on the associated data lines 22. Then, the sending processor 12 will release the write line 20 and the receiving processor (12e in this example) pulls both the write line 20 and the read line 18 low. The latter action will acknowledge to the sending processor 12 that the data has been received. Note that the above description is not intended necessarily to denote the sequence of events in order. In actual practice, the receiving processor may try to set the write line 20 low slightly before the sending processor 12 releases (stops pulling high) its write line 20. In such an instance, as soon as the sending processor 12 releases its write line 20, the write line 20 will be pulled low by the receiving computer 12e. It is important to note that when the write line 20 of the sending processor 12 goes high, the data or code is already transferred; therefore, the receiving processor (12e in this instance) merely needs to latch the data/code for an essentially instantaneous response.

[0029] If the processor 12e were attempting to write to the processor 12a, then processor 12e would set the write line 20 between processor 12e and processor 12a to high. If the read line 18 between processor 12e and processor 12a has then not already been set to high by processor 12a, then processor 12e will simply wait until processor 12a does set that read line 20 high. When both of a corresponding pair of write line 18 and read line 20 are set high, then the data awaiting to be transferred on the data lines 22 is transferred. Thereafter, the receiving processor 12 (processor 12a, in this example) sets both the read line 18 and the write line 20 between the two processors (12e and 12a in this example) to low as soon as the sending processor 12e releases the write line 18.

[0030] Whenever a processor 12 such as the processor 12e has set one of its write lines 20 high in anticipation of writing it will simply wait, using essentially no power, until the data is "requested", as described above, from the appropriate adjacent processor 12, unless the processor 12 to which the data is to be sent has already set its read line 18 high, in which case the data is transmitted immediately. Similarly, whenever a processor 12 has set one or more of its read lines 18 to high in anticipation of reading, it will simply wait, using essentially no power, until the write line 20 connected to a selected processor 12 goes high to transfer an instruction word between the two processors 12.

[0031] As discussed above, there may be several potential means and/or methods to cause the processors 12 to function as described. However, in this present example, the processors 12 so behave simply because they are operating generally asynchronously internally (in addition to transferring data there-between in the asynchronous manner described). That is, instructions are generally completed sequentially. When either a write or read instruction occurs, there can be no further action until that instruction is completed (or, perhaps alternatively, until it is aborted, as by a "reset" or the like). There is no regular clock pulse, in the prior art sense. Rather, a pulse is generated to accomplish a next instruction only when the instruction being executed either is not a read or write type instruction (given that a read or write type instruction would require completion, often by another entity) or else when the read or write type operation is, in fact, completed.

[0032] Fig. 3 is a block diagram depicting the general layout of an example of one of the processors 12 of Figs. 1 and 2. As can be seen in the view of Fig. 3, each of the processors 12 is a generally self contained computer having its own RAM 24 and ROM 26. As mentioned previously, the processors 12 are also sometimes referred to as individual "nodes", given that they are, in the present example, combined on a single chip.

[0033] Other basic components of the processor 12 are a return stack 28 (including an R register 29, discussed hereinafter), an instruction area 30, an arithmetic logic unit ("ALU" or "processor") 32, a data stack 34, a decode logic section 36 for decoding instructions, and a slot sequencer 42. One skilled in the art will be generally familiar with the operation of stack based computers such as the processors 12 of this present example. The processors 12 are dual stack processors having the data stack 34 and the separate return stack 28. Fig. 3 also shows circular register arrays 28a and 34a for the return stack and data stack, respectively, along with the T register 44 and S register 46 of the data stack 34.

[0034] In this embodiment of the invention, the processor 12 has four communication ports 38 for communicating with adjacent processors 12. The communication ports 38 are tri-state drivers, having an off status, a receive status (for driving signals into the processor 12) and a send status (for driving signals out of the processor 12). If the particular processor 12 is not on the interior of the array (Fig. 1) such as the example of processor 12e,

then one or more of the communication ports 38 will not be used in that particular processor, at least for the purposes described above. However, those communication ports 38 that do abut the edge of the die 14 can have additional circuitry, either designed into such processor 12 or else external to the processor 12 but associated therewith, to cause such communication port 38 to act as an external I/O port 39 (Fig. 1). Examples of such external I/O ports 39 include, but are not limited to, USB (universal serial bus) ports, RS232 serial bus ports, parallel communications ports, analog to digital and/or digital to analog conversion ports, and many other possible variations. No matter what type of additional or modified circuitry is employed for this purpose, according to the presently described embodiment of the invention, the method of operation of the "external" I/O ports 39 regarding the handling of instructions and/or data received there from will be alike to that described herein, in relation to the "internal" communication ports 38. In Fig. 1 an "edge" processor 12f is depicted with associated interface circuitry 80 (shown in block diagrammatic form) for communicating through an external I/O port 39 with an external device 82.

[0035] In the presently described embodiment, the instruction area 30 includes a number of registers 40 including, in this example, an A register 40a, a B register 40b and a P register 40c. In this example, the A register 40a is a full eighteen-bit register, while the B register 40b and the P register 40c are nine-bit registers. An I/O register 47 (18 bit) is located between the memory (ROM 26 and RAM 24) and the communication ports 38. The I/O register 47 will be disclosed in greater detail hereinafter.

[0036] Although the invention is not limited by this example, the present processor 12 is implemented to execute native Forth language instructions. As one familiar with the Forth computer language will appreciate, complicated Forth instructions, known as Forth "words" are constructed from the native processor instructions designed into the processor. The collection of Forth words is known as a "dictionary". As will be described in greater detail hereinafter, the processor 12 reads eighteen bits at a time from RAM 24, ROM 26 or directly from one of the data buses 16 (Fig. 2). However, since in Forth most instructions (known as operand-less instructions) obtain their operands directly from the stacks 28 and 34, they are generally only five bits in length, such that up to four instructions can be included in a single eighteen-bit instruction word, with the condition that the last instruction in the group is selected from a limited set of instructions that require only three bits.

[0037] Fig. 4 is a diagrammatic representation of an instruction word 48. (It should be noted that the instruction word 48 can actually contain instructions, data, or some combination thereof.) The instruction word 48 consists of eighteen bits 50. This being a binary computer, each of the bits 50 will be a '1' or a '0'. As previously discussed herein, the eighteen-bit wide instruction word 48 can contain up to four instructions 52 in four slots 54 called slot zero 54a, slot one 54b, slot two 54c and slot three 54d. In the present embodiment of the invention, the eighteen-bit instruction words 48 are always read as a whole. Therefore, since there is always a potential of having up to four instructions in the instruction word 48, a NOP (no operation) instruction is included in the instruction set of the processor 12 to provide for instances when using all of the available slots 54 might be unnecessary or even undesirable.

[0038] In addition to the registers previously discussed herein, the instruction area 30 also has an 18 bit instruction register 30a for storing the instruction word 48 that is presently being used, and an additional 5 bit opcode register 30b for the instruction in the particular instruction word presently being executed.

[0039] As discussed above, the i4 bit 66 of each instruction 52 is set according to whether or not that instruction is either of a read or a write type of instruction, as opposed to that instruction being one that requires no input or output. The remaining bits 50 in the instruction 52 provide the remainder of the particular opcode for that instruction. In the case of a read or write type instruction, one or more of the bits may be used to indicate where data is to be read from, or written to, in that particular processor 12. In the present example of the invention, data to be written always comes from the T register 44 (the top of the data stack 34); however data can be selectively read into either the T register 44 or else the instruction area 30 from where it can be executed. In this particular embodiment of the invention, either data or instructions can be communicated in the manner described herein and instructions can therefore, be executed directly from the data bus 16.

[0040] One or more of the bits 50 will be used to indicate which of the ports 38, if any, is to be set to read or write. This later operation is optionally accomplished by using one or more bits to designate a register 40, such as the A register 40a, the B register 40b, or the like. In such an example, the designated register 40 will be preloaded with data having a bit corresponding to each of the ports 38 (and, also, any other potential entity with which the processor 12 may be attempting to communicate, such as memory (RAM 24 or ROM 26), an external communications port 39, or the like.) For example, each of four bits in the particular register 40 can correspond to each of the up port 38a, the right port 38b, the left port 38c or the down port 38d. In such case, where there is a '1' at any of those bit locations, communication will be set to proceed through the corresponding port 38. In the present embodiment of the invention, it is anticipated that a read opcode might set more than one port 38 for communication in a single instruction.

[0041] The immediately following example will assume a communication wherein processor 12e is attempting to write to processor 12c, although the example is applicable to communication between any adjacent processors 12. When a write instruction is executed in a writing processor 12e, the selected write line 20 (in this example,

the write line 20 between processors 12e and 12c) is set high; if the corresponding read line 18 is already high then data is immediately sent from the selected location through the selected communications port 38. Alternatively, if the corresponding read line 18 is not already high, then processor 12e will simply stop operation until the corresponding read line 18 does go high.

[0042] As for how the operation of the processor 12e is resumed when a read or write type instruction is completed, the mechanism for that is as follows. When both the read line 18 and the corresponding write line 20 between processors 12e and 12c are high, then both lines 18 and 20 will be released by each of the respective processors 12 that is holding it high. (In this example, the sending processor 12e will be holding the write line 18 high while the receiving processor 12c will be holding the read line 20 high). Then the receiving processor 12c will pull both lines 18 and 20 low. In actual practice, the receiving processor 12c may attempt to pull the lines 18 and 20 low before the sending processor 12e has released the write line 18. However, since the lines 18 and 20 are pulled high and only weakly held (latched) low, any attempt to pull a line 18 or 20 low will not actually succeed until that line 18 or 20 is released by the processor 12 that is holding it high.

[0043] When both lines 18 and 20 in a data bus 16 are pulled low, this is an "acknowledge" condition. Each of the processors 12e and 12c will, upon the acknowledge condition, set its own internal acknowledge line high.

[0044] As can be appreciated in light of the above discussion, the operations are essentially the same whether processor 12e attempts to write to processor 12c first, or whether processor 12c first attempts to read from processor 12e. The operation cannot be completed until both processors 12e and 12c are ready and, whichever processor 12e or 12c is ready first, that first processor 12 simply "becomes inactive" until the other processor 12e or 12c completes the transfer. Another way of looking at the above described process is that, actually, both the writing processor 12e and the receiving processor 12c become inactive when they execute the write and read instructions, respectively, but the last one to enter into the transaction reactivates nearly instantaneously when both the read line 18 and the write line 20 are high, whereas the first processor 12 to initiate the transaction can stay inactive nearly indefinitely until the second processor 12 is ready to complete the process.

[0045] The inventor believes that a key feature for enabling efficient asynchronous communications between devices is a type of acknowledge signal or condition. In the prior art, most communication between devices has been clocked and there is no direct way for a sending device to know that the receiving device has properly received the data. Methods such as checksum operations may have been used to attempt to insure that data is correctly received, but the sending device has no direct indication that the operation is completed. The present inventive method, as described herein, provides the necessary acknowledge condition that allows, or at least makes practical, asynchronous communications between the devices. Furthermore, the acknowledge condition also makes it possible for one or more of the devices to "become inactive" until the acknowledge condition occurs. An acknowledge condition could be communicated between the processors 12 by a separate signal being sent between the processors 12 (either over the interconnecting data bus 16 or over a separate signal line), and such an acknowledge signal would be within the scope of this aspect of the present invention. However, according to the embodiment of the invention described herein, it can be appreciated that there is even more economy involved here, in that the method for acknowledgement does not require any additional signal, clock cycle, timing pulse, or any such resource beyond that described, to actually affect the communication.

[0046] Since four instructions 52 can be included in an instruction word 48 and since, according to the present invention, an entire instruction word 48 can be communicated at one time between processors 12, this presents an ideal opportunity for transmitting a very small program in one operation. For example most of a small "For/Next" loop can be implemented in a single instruction word 48. Fig. 5 is a diagrammatic representation of a micro-loop 100. The micro-loop 100, not unlike other prior art loops, has a FOR instruction 102 and a NEXT instruction 104. Since an instruction word 48 (Fig. 4) contains as many as four instructions 52, an instruction word 48 can include three operation instructions 106 within a single instruction word 48. The operation instructions 106 can be essentially any of the available instructions that a programmer might want to include in the micro-loop 100. A typical example of a micro-loop 100 that might be transmitted from one processor 12 to another might be a set of instructions for reading from, or writing to the RAM 24 of the second processor 12, such that the first processor 12 could "borrow" available RAM 24 capacity.

[0047] The FOR instruction 102 pushes a value onto the return stack 28 representing the number of iterations desired. That is, the value on the T register 44 at the top of the data stack 34 is pushed onto the R register 29 of the return stack 28. The FOR instruction 102 can be located in any slot 54. Where the FOR instruction 102 is not located in slot three 54d, then the remaining instructions 52 in that instruction word 48 will be executed before going on to the micro-loop 100, which will generally be the next loaded instruction word 48.

[0048] According to the presently described embodiment of the invention, the NEXT instruction 104 depicted in the view of Fig. 5 is a particular type of NEXT instruction 104. This is because it is located in slot three 54d (Fig. 4) According to this embodiment of the invention, it is assumed that all of the data in a particular instruction word 48 that follows an "ordinary" NEXT instruction (not shown) is an address (the address where the for/next loop begins). The opcode for the NEXT instruction 104 is the same, no matter which of the four slots 54 it is in

(with the obvious exception that the last two digits are assumed if it is in slot three 54d, rather than being explicitly written, as discussed previously herein). However, since there can be no address data following the NEXT instruction 104 when it is in slot three 54d, it can be also assumed that the NEXT instruction 104 in slot three 54d is a MICRO-NEXT instruction 104a. The MICRO-NEXT instruction 104a uses the address of the first instruction 52, located in slot zero 54a of the same instruction word 48 in which it is located, as the address to which to return. The MICRO-NEXT instruction 104a also takes the value from the R register 29 (which was originally pushed there by the FOR instruction 102), decrements it by 1, and then returns it to the R register 29. When the value on the R register 29 reaches a predetermined value (such as zero), then the MICRO-NEXT instruction will load the next instruction word 48 and continue on as described previously herein. However, when the MICRO-NEXT instruction 104a reads a value from the R register 29 that is greater than the predetermined value, it will resume operation at slot zero 54a of its own instruction word 48 and execute the three instructions 52 located in slots zero through three, inclusive, thereof. That is, a MICRO-NEXT instruction 104a will always, in this embodiment of the invention, execute three operation instructions 106. Because, in some instances, it may not be desired to use all three potentially available instructions 52, a "NOP" instruction is available to fill one or two of the slots 54, as required.

[0049] It should be noted that micro-loops 100 can be used entirely within a single processor 12. Indeed, the entire set of available machine language instructions is available for use as the operation instructions 106, and the application and use of micro-loops is limited only by the imagination of the programmer. However, when the ability to execute an entire micro-loop 100 within a single instruction word 48 is combined with the ability to allow a processor 12 to send the instruction word 48 to a neighbor processor 12 to execute the instructions 52 therein essentially directly from the data bus 16, this provides a powerful tool for allowing a processor 12 to utilize the resources of its neighbors.

[0050] The small micro-loop 100, all contained within the single instruction word 48, can be communicated between processors 12, as described herein and it can be executed directly from the communications port 38 of the receiving processor 12, just like any other set of instructions contained in an instruction word 48, as described herein. While there are many uses for this sort of "micro-loop" 100, a typical use would be where one processor 12 wants to store some data onto the memory of a neighbor processor 12. It could, for example, first send an instruction to that neighbor processor telling it to store an incoming data word to a particular memory address, then increment that address, then repeat for a given number of iterations (the number of data words to be transmitted). To read the data back, the first processor would just instruct the second processor (the one used for storage here) to write the stored data back to the first processor, using a similar micro-loop.

[0051] By using the micro-loop 100 structure in conjunction with the direct execution aspect described herein, a processor 12 can use an otherwise resting neighbor processor 12 for storage of excess data when the data storage need exceeds the relatively small capacity built into each individual processor 12. While this example has been described in terms of data storage, the same technique can equally be used to allow a processor 12 to have its neighbor share its computational resources - by creating a micro-loop 100 that causes the other processor 12 to perform some operations, store the result, and repeat a given number of times. As can be appreciated, the number of ways in which this inventive micro-loop 100 structure can be used is nearly infinite.

[0052] As previously mentioned herein, in the presently described embodiment of the invention, either data or instructions can be communicated in the manner described herein and instructions can, therefore, be executed essentially directly from the data bus 16. That is, there is no need to store instructions to RAM 24 and then recall them before execution. Instead, according to this aspect of the invention, an instruction word 48 that is received on a communications port 38 is not treated essentially differently than it would be were it recalled from RAM 24 or ROM 26.

[0053] One of the available machine language instructions is a FETCH instruction. The FETCH instruction uses the address on the A register 40a to determine from where to fetch an 18 bit word. Of course, the program will have to have already provided for placing the correct address on the A register 40a. As previously discussed herein, the A register 40a is an 18 bit register, such that there is a sufficient range of address data available that any of the potential sources from which a fetch can occur can be differentiated. That is, there is a range of addresses assigned to ROM, a different range of addresses assigned to RAM, and there are specific addresses for each of the ports 38 and for the external I/O port 39. A FETCH instruction always places the 18 bits that it fetches on the T register 44. An important advantage exists here in that there are no instruction fetches inside the loop. Therefore, there is approximately a 30% increase in efficiency, with a corresponding decrease in power consumption.

[0054] In contrast, as previously discussed herein, executable instructions (as opposed to data) are temporarily stored in the instruction register 30a. There is no specific command for "fetching" an 18 bit instruction word 48 into the instruction register 30a. Instead, when there are no more executable instructions left in the instruction register 30a, then the processor will automatically fetch the "next" instruction word 48. Where that "next" instruction word is located is determined by the "program counter" or "pc", herein called the P register 40c. The P register 40c is often automatically incremented, as is the case where a sequence of instruction words 48 is to be fetched from RAM 24 or ROM 26. However, there are a number

of exceptions to this general rule. For example, a JUMP or CALL instruction will cause the P register 40c to be loaded with the address designated by the data in the remainder of the presently loaded instruction word 48 after the JUMP or CALL instruction, rather than being incremented. When the P register 40c is then loaded with an address corresponding to one or more of the ports 38, then the next instruction word 48 will be loaded into the instruction register 30a from the ports 38. The P register 40c also does not increment when an instruction word 48 has just been retrieved from a port 38 into the instruction register 30a. Rather, it will continue to retain that same port address until a specific JUMP or CALL instruction is executed to change the P register 40c. That is, once the processor 12 is told to look for its next instruction from a port 38, it will continue to look for instructions from that same port 38 (or ports 38) until it is told to look elsewhere, such as back to the memory (RAM 24 or ROM 26) for its next instruction word 48.

[0055] As noted above, the processor 12 knows that the next eighteen bits fetched is to be placed in the instruction register 30a when there are no more executable instructions left in the present instruction word 48. By default, there are no more executable instructions left in the present instruction word 48 after a JUMP or CALL instruction (or also after certain other instructions that will not be specifically discussed here) because, by definition, the remainder of the 18 bit instruction word following a JUMP or CALL instruction is dedicated to the address referred to by the JUMP or CALL instruction. Another way of stating this is that the above described processes are unique in many ways, including but not limited to the fact that a JUMP or CALL instruction can, optionally, be to a port 38, rather than to just a memory address, or the like.

[0056] It should be remembered that, as discussed previously herein, the processor 12 can look for its next instruction from one port 38 or from any of a group of the ports 38. Therefore, addresses are provided to correspond to various combinations of the ports 38. When, for example, a processor is told to FETCH an instruction from a group of ports 38, then it will accept the first available instruction word 48 from any of the selected ports 38. If no neighbor processor 12 has already attempted to write to any of those ports 38, then the processor 12 in question will "become inactive", as described in detail above, until a neighbor does write to the selected port 38.

[0057] Fig. 6 is a flow diagram depicting an example of the above described direct execution method 120. A "normal" flow of operations will commence when, as discussed previously herein, there are no more executable instructions left in the instruction register 30a. At such time, the processor 12 will "fetch" another instruction word (note that the term "fetch" is used here in a general sense, in that an actual FETCH instruction is not used), as indicated by a "fetch word" operation 122. That operation will be accomplished according to the address in the P register 40c (as indicated by an "address" decision operation 124 in the flow diagram of Fig. 6). If the address in the P register 40c is a RAM 24 or ROM 26 address, then the next instruction word 48 will be retrieved from the designated memory location in a "fetch from memory" operation 126. If, on the other hand, the address in the P register 40c is that of a port 38 or ports 38 (not a memory address), then the next instruction word 48 will be retrieved from the designated port location in a "fetch from port" operation 128. In either case, the instruction word 48 being retrieved is placed in the instruction register 30a in a "retrieve instruction word" operation 130. In an "execute instruction word" operation 132, the instructions in the slots 54 of the instruction word 48 are accomplished sequentially, as described previously herein.

[0058] In a "jump" decision operation 134, it is determined if one of the operations in the instruction word 48 is a JUMP instruction, or other instruction that would divert operation away from the continued "normal" progression as discussed previously herein. If yes, then the address provided in the instruction word 48 after the JUMP (or other such) instruction is provided to the P register 40c in a "load P register" operation 136, and the sequence begins again in the "fetch word" operation 122, as indicated in the diagram of Fig. 6. If no, then the next action depends upon whether the last instruction FETCH was from a port 38 or from a memory address, as indicated in a "port address" decision operation 138. If the last instruction FETCH was from a port 38, then no change is made to the P register 30a and the sequence is repeated starting with the "fetch word" operation 122. If, on the other hand, the last instruction FETCH was from a memory address (RAM 24 or ROM 26), then the address in the P register 30a is incremented, as indicated by an "increment P register" operation 140 in Fig. 6, before the "fetch word" operation 122 is accomplished.

[0059] The above description is not intended to represent actual operational steps. Instead, it is a diagram of the various decisions and operations resulting there from that are performed according to the described embodiment of the invention. Indeed, this flow diagram should not be understood to mean that each operation described and shown requires a separate distinct sequential step. In fact many of the described operations in the flow diagram of Fig. 6 will, in practice, be accomplished generally simultaneously.

[0060] Fig. 7 is a flow diagram depicting an example of a method for alerting a processor 150. As previously discussed herein, the processors 12 of the embodiment described will "become inactive" while awaiting an input. Such an input can be from a neighboring processor 12, as in the embodiment described in relation to Figs. 1 through 4. Alternatively, as was also discussed previously herein, the processors 12 that have communication ports 38 that abut the edge of the die 14 can have additional circuitry, either designed into such processor 12 or else external to the processor 12 but associated therewith, to cause such communication port 38 to act as an external I/O port 39. In either case, the inventive combi-

nation can provide the additional advantage that the "inactive" processor 12 can be poised and ready to activate and spring into some prescribed action when an input is received. This process is referred to as a worker mode.

[0061] Each processor 12 is programmed to JUMP to an address when it is started. That address will be the address of the first instruction word 48 that will start that particular processor 12 on its designated job. The instruction word can be located, for example, in the ROM 26. After a cold start, a processor 12 may load a program, such as a program known as a worker mode loop. The worker mode loop for center processors 12, edge processors 12, and corner processors 12 will be different. In addition, some processors 12 may have specific tasks at boot-up in ROM associated with their positions within the array 10. Worker mode loops will be described in greater detail hereinafter.

[0062] While there are numerous ways in which this feature might be used, an example that will serve to illustrate just one such "computer alert method" is illustrated in the view of Fig. 7 and is enumerated therein by the reference character 150. As can be seen in the view of Fig. 7 in an "inactive but alert state" operation 152, a processor 12 is caused to "become inactive" such that it is awaiting input from a neighbor processor 12, or more than one (as many as all four) neighbor processors or, in the case of an "edge" processor 12 an external input, or some combination of external inputs and/or inputs from a neighbor processor 12. As described previously herein, a processor 12, can "become inactive" awaiting completion of either a read or a write operation. Where the processor 12 is being used, as described in this example, to await some possible "input", then it would be natural to assume that the waiting processor has set its read line 18 high awaiting a "write" from the neighbor or outside source. Indeed, it is presently anticipated that will be the usual condition. However, it is within the scope of the invention that the waiting processor 12 will have set its write line 20 high and, therefore, that it will become activated when the neighbor or outside source "reads" from it.

[0063] In an "activate" operation 154, the inactive processor 12 is caused to resume operation because the neighboring processor 12 or external device 39 has completed the transaction being awaited. If the transaction being awaited was the receipt of an instruction word 48 to be executed, then the processor 12 will proceed to execute the instructions therein. If the transaction being awaited was the receipt of data, then the processor 12 will proceed to execute the next instruction in queue, which will be either the instruction in the next slot 54 in the present instruction word 48, or else the next instruction word 48 will be loaded and the next instruction will be in slot 0 of that next instruction word 48. In any case, while being used in the described manner, then that next instruction will begin a sequence of one or more instructions for handling the input just received. Options for handling such input can include reacting to perform some predefined function internally, communicating with one or more of the other processors 12 in the array 10, or even ignoring the input (just as conventional prior art interrupts may be ignored under prescribed conditions). The options are depicted in the view of Fig. 7 as an "act on input" operation 156. It should be noted that, in some instances, the content of the input may not be important. In some cases, for example, it may be only the very fact that an external device has attempted communication that is of interest.

[0064] One skilled in the art will recognize that this above described operating mode will be useful as a more efficient alternative to the conventional use of interrupts. When a processor 12 has one or more of its read lines 18 (or a write line 20) set high, it can be said to be in an "alert" condition. In the alert condition, the processor 12 is ready to immediately execute any instruction sent to it on the data bus 16 corresponding to the read line or lines 18 that are set high or, alternatively, to act on data that is transferred over the data bus 16. Where there is an array of processors 12 available, one or more can be used, at any given time, to be in the above described alert condition such that any of a prescribed set of inputs will trigger it into action. This is preferable to using the conventional interrupt technique to "get the attention" of a processor, because an interrupt will cause a processor to have to store certain data, load certain data, and so on, in response to the interrupt request. According to the present invention, a processor can be placed in the alert condition and dedicated to awaiting the input of interest, such that not a single instruction period is wasted in beginning execution of the instructions triggered by such input. Again, note that in the presently described embodiment, processors in the alert condition will actually be "inactive", meaning that they are using essentially no power, but "alert" in that they will be instantly triggered into action by an input. However, it is within the scope of this aspect of the invention that the "alert" condition could be embodied in a processor even if it were not "inactive". The described alert condition can be used in essentially any situation where a conventional prior art interrupt (either a hardware interrupt or a software interrupt) might have otherwise been used.

[0065] Fig. 8 is another example of a processor alert method 150a. This is but one example wherein interaction between a monitoring processor 12f (Fig. 1) and another processor 12g (Fig. 1) that is assigned to some other task may be desirable or necessary. As can be seen in the view of Fig. 8, there are two generally independent flow charts, one for each of the processors 12f and 12g. This is indicative of the nature of the cooperative coprocessor approach of the present invention, wherein each of the processors 12 has its own assignment which it carries out generally independently, except for those occasions when interaction is accomplished as described herein. This invention provides an alternative to the use of interrupts to handle inputs, whether such inputs come from an external input device, or from another proc-

essor 12 in the array 10. Instead of causing a processor 12 to have to stop what it is doing in order to handle an interrupt, the inventive combination described herein will allow for a processor 12 to be in an "inactive but alert" state, as previously described. Therefore, one or more processors 12 can be assigned to receive and act upon certain inputs.

[0066] Regarding the processor 12f and the "enter inactive but alert status" operation 152, the "activate" operation 154, and the "act on input" operation 156 each are accomplished as described previously herein in relation to the first example of the processor alert method 150. However, because this example anticipates a possible need for interaction between the processors 12f and 12g, then following the "act on input" operation 156, the processor 12f enters a "send info?" decision operation 158 wherein, according to its programming, it is determined if the input just received requires the attention of the other processor 12g. If no, then the processor 12f returns to inactive but alert status, or some other alternative such as was discussed previously herein. If yes, then the processor 12f initiates communication with the processor 12g as described in detail previously herein in a "send to other" operation 160. It should be noted that, according to the choice of the programmer, the processor 12f could be sending instructions such as it may have generated internally in response to the input from the external device 82. Alternatively, the processor 12f could pass on data to the processor 12g and such data could be internally generated in processor 12 or else "passed through" from the external device 82. Still another alternative might be that the processor 12f, in some situations, might attempt to read from the processor 12g when it receives an input from the external device 82. All of these opportunities are available to the programmer.

[0067] The processor 12g is generally executing code to accomplish its assigned primary task, whatever that might be, as indicated in an "execute primary function" operation 162. However, if the programmer has decided that occasional interaction between the processors 12f and 12g is desirable, then the programmer will have provided that the processor 12g occasionally pause to see if one or more of its neighbors has attempted a communication, as indicated in a "look for input" operation 166. An "input?" decision operation 168 is made in the event that there is a communication waiting (as, for example, if the processor 12f has already initiated a write to the processor 12g). If there has been a communication initiated (yes), then the processor 12g will complete the communication, as described in detail previously herein, in a "receive from other" operation 170. If no, then the processor 12g will return to the execution of its assigned function, as shown in Fig. 8. After the "receive from other" operation 170, the processor 12g will act on the input received in an "act on input" operation 172. As mentioned above, the programmer could have provided that the processor 12g would be expecting instructions such as input, in which case the processor 12g would execute the instructions as described previously herein. Alternatively, the processor 12g might be programmed to be expecting data to act upon.

[0068] In the example of Fig. 8, it is shown that following the "act on input" operation 172, the processor 12g returns to the accomplishment of its primary function (that is, it returns to the "execute primary function" operation 162). However the possibility of even more complicated examples certainly exists. For instance, the programming might be such that certain inputs received from the processor 12f will cause it to abort its previously assigned primary function and begin a new one, or else it might simply temporarily stop and await further input. As one skilled in the art will recognize, the various possibilities for action here are limited only by the imagination of the programmer.

[0069] It should be noted that, according to the embodiment of the invention described herein, a given processor 12 need not be interrupted while it is performing a task because another processor 12 is assigned the task of monitoring and handling inputs that might otherwise require an interrupt. However, it is interesting to note also that the processor 12 that is busy handling another task also cannot be disturbed unless and until its programming provides that it look to its ports 38 for input. Therefore, it will sometimes be desirable to cause the processor 12 to pause or suspend its current task to look for other inputs. The concept of "Pause" and how it is used will be discussed in greater detail later.

[0070] Each port 38 between processors 12 comprises data lines 22 and one read line 18 and one write line 20, which inclusively make up the data bus 16. In addition to the data bus 16, each port 38 also comprises handshake control signals. The data lines 22 connect between the ports 38 of two adjacent processors 12. For example, a word or op code may reside in the T register 44 of processor 12e during a STORE (write) instruction; the write line 20 of processor 12e would then be set high. When the read line 18 of processor 12c is set high, then data is transferred into the T register 44 of processor 12c in a FETCH (read) instruction. After the transaction is complete, both the read line 18 and the write line 20 are set low. In this example, this data becomes an instruction when it is read by the P register 40c.

[0071] When a processor 12 reads a message, the message could be in the form of data, instructions, or signals. Instructions may be stored into memory and used later by the same processor 12, or stored into and executed directly from a port 38 by a different processor 12. If a processor 12 is reading from memory using its P register 40c, it will either immediately execute the instruction by putting the instruction into the instruction register 30a, or it will read the message as data and put it into the T register 44. A FETCH instruction will read a message as data when the FETCH instruction is directed or addressed to a port 38. If a JUMP or CALL instruction is to a port 38, or a RETURN instruction is to a port 38 address, then the P register 40c will read what is written

to the port 38 as instructions and the instruction will be treated as executable code.

[0072] A receiving processor 12 could read a message as data and then write a message as data. A message that is routed (i.e. sent from one processor 12 to a non-adjacent processor 12 through intermediary processors 12) is interpreted as data and read into the T register 44 of each successive processor 12 until the intended recipient is reached, then the message is interpreted as code (read from the P register 40c) and then executed. Therefore, if a message is read during FETCH A (defined as reading the contents of memory specified by the A register 40a) or FETCH A+ (defined as reading the contents of memory specified by the A register 40a, and adding one to the A register 40a) or FETCH P+ (defined as reading the contents specified by the P register 40c, and adding one to the P register 40c), then the message is transferred into the T register 44 of the processor 12 doing the reading. If the processor 12 is reading the message from the P register 40c, then the message is transferred into the instruction register 30a of the receiving processor 12.

[0073] Fig. 1 showed an array 10 of interconnected processors 12 located on a single die 14; a total of 24 processors 12 were given as an example, wherein each processor 12 has several pins located about its periphery. Each processor 12 has four ports 38 that are designated as right, down, left, and up (RDLU). In Fig. 1, processor 12e has four adjacent processors 12, wherein processor 12b is the right neighbor, processor 12d is the down neighbor, processor 12c is the left neighbor, and processor 12a is the up neighbor, all with respect to the center processor 12e. Even though the edge processors 12 have only three adjacent neighbors and the corner processors 12 have only two adjacent neighbors, these edge and corner processors 12 still have four ports 38 which are also designated as RDLU.

[0074] Fig. 9 is a block diagrammatic depiction of an alternative array 10a. In this example of the invention, the array 10a has twenty four (24) processors 12. Also, in this embodiment of the invention, the processors 12 are arranged in a particular relative orientation referred to as "mirroring". That is, the processors 12 in the second and fourth rows 173 from the top of the array 10a have been flipped about their x-axes 174, so that the down ports 38d are now facing upward. All of the processors 12 in the second, fourth, and sixth columns 176 with respect to the left side of the array 10a have been flipped about their y-axes 178, such that the right ports 38b are now facing towards the left side of the array 10. This results in processors N6, N8, N10, N18, N20 and N22 maintaining their original RDLU orientations; processors N0, N2, N4, N12, N14, and N16 having been flipped about their x-axes 174 only; processors N7, N9, N11, N19, N21, and N23 having been flipped about their y-axes 178 only; and nodes N1, N3, N5, N13, N15, and N17 having been flipped about both their x-axes 174 and their y-axes 178. With the exception of the processors 12 located at the corners and edges of the array 10a, these rotations result in all of the right ports 38b directly facing each other; all of the down ports 38d directly facing each other; all of the left ports 38c directly facing each other; and all of the up ports 38a directly facing each other. As will be discussed in greater detail hereinafter, this allows the processors 12 to directly align and connect with their nearest neighbor processors 12 because the interconnects of a processor 12 "mirror" the interconnects of an adjacent connecting neighbor processor 12.

[0075] In order to have a way to refer to directions within the array 10a that does not change according to which way the processors 12 therein are mirrored, the inventors have chosen to use the terminology North, South, East and West (NSEW) The directions of North, South, East, and West maintain their relative directions, even with mirroring. This is relevant during routing, which was previously described as sending a message from a processor 12 to another non-adjacent processor 12 through intermediary processors 12. Directions (NSEW) are in a table located in ROM 26.

[0076] Left ports 38c and up ports 38a do not connect to anything internal to the array 10a when they are on the outside border of the array 10a although, as discussed previously herein, they will probably connect to an external I/O port 39 (Fig. 1). Down ports 38d and right ports 38b always connect to another processor 12 so long as the number of rows and columns is an even number. As an example, processor N7 has four orthogonally adjacent neighbors, namely N6 which is connected to right port 38b, N1 which is connected to down port 38d, N8 which is connected to left port 38c, and N13 which is connected to the up port 38a. Fig. 9a is an expanded view of four nodes, namely N7, N8, N13, and N14 from Fig. 9, with right port 38b, down port 38d, left port 38c, and up port 38a notations.

[0077] Each processor 12 has an 18 bit I/O register 47, as shown in Fig. 3. Each I/O register 47 contains, inter alia, information as to whether an adjacent neighbor is reading from or writing to its ports 38. Fig. 10 is a partial view of an I/O register 47 for a processor 12. Bits B9 through B16 indicate the read and write status for a processor 12. The I/O register 47 contains the read and write handshake status bits 50 on its communication ports 38. These are read-only bits 50. By reading these, a processor 12 can see if a neighbor is inactive waiting to write to one of its ports 38 or is inactive waiting to read from one of its ports 38. If a neighbor is waiting to write to a processor 12, then the I/O register 47 write line status bit of processor 12 will go high, indicating that a write message from that particular neighbor is waiting to be sent. Likewise, if a neighbor is waiting to read from processor 12, then the I/O register 47 read line status bit of processor 12 will go high, indicating that a read message for that particular neighbor is waiting to be received.

[0078] The following example with reference to Figs. 9, 9a, and 10 will further exemplify the above procedure. The I/O register 47 for Node 7 indicates the read and

write status bits (B16 and B15, respectively) for the right port 38b, which is connected to processor N6 in this example. Bits B14 and B13 are the read and write status bits, respectively, for the down port 38d (connected to processor N1); bits B12 and B11 are the read and write status bits, respectively, for the left port 38c (connected to processor N8); and bits N10 and N9 are the read and write status bits, respectively, for the up port 38a (connected to processor N13). In this example, bits 16 - 9 always give the read and write status of adjacent nodes in the order of right, down, left, and up (RDLU). Another explanation of the I/O register 47 is as follows with continued reference to Fig. 10, which shows a partial I/O register for Node 7. If bit B16 is high, then there is a read request from processor N6; if bit B15 is high, then there is a write request from processor N6; if bit B14 is high, then there is a read request from processor N1; and so forth.

[0079] As discussed earlier, PAUSE causes a processor 12 to temporarily suspend its processing tasks or remain inactive in order to check for incoming data or instructions. There are two instances for using a PAUSE routine. The first instance occurs after a processor 12 becomes activated from a previous inactive state. The second instance occurs when a processor 12 is executing a program but takes a break, or pause, to look for an incoming message.

[0080] A NOP (also referred to as a "no-op") is a no operation instruction, and is designated by four dots (....) in the instruction code. With reference back to Fig. 4, a NOP may be used in the instance when using some or all of the available slots 54 might be unnecessary or undesirable. For example, in a message, four NOPs (....) are used as a message header, in part because it must be safe to throw away the header (as after being activated) and also safe to execute the header (as when already executing or taking a break from executing the processor's own code), both of which conditions are met by a NOP instruction. It also has to be safe for multiple messages to arrive from different directions generally simultaneously. When each message begins with four NOPs, a simultaneous reading of two or more different messages will not be catastrophic since each processor is reading the same message header.

[0081] A processor 12 can be designated as primarily a worker or "production type" processor 12. In the absence of any other instructions, this processor 12 can become a worker by default, and perform a worker mode loop 200 as in Fig. 11, the loop being located in ROM. This worker processor 12 remains asleep or inactive until a message header containing four NOPS is sent to the worker processor 12 at the start of the worker mode loop 200. Fig. 11 is a flow diagram representing an example of a worker mode loop 200 which utilizes a PAUSE routine. When a worker processor 12 is inactive in a default worker mode loop 200 in ROM, the four NOPs of the message header are read as data in a "read message" operation 210. When the message arrives, a FETCH A instruction reads one word as data, which is four NOPs from a neighbor processor 12, which is placed into the T register 44 of the worker processor 12. Reading these four NOPS will wake up the worker processor 12, as indicated by a "wake up" operation 211 in the flow chart of Fig. 11. The "wake-up" message will activate the worker processor 12. As part of the worker mode loop 200, the contents or addresses of the B register 40b are pointed at the I/O register 47 as a default setting, and therefore a FETCH B instruction will read the contents of the I/O register 47 in a "read I/O register" operation 212 to determine from which port the message was sent.

[0082] In a "begin pause" operation 213, the worker processor 12 is absent of any processing activities, in order to prepare for the next step of "check the appropriate port(s)" operation 214, according to the contents of the I/O register 47 that were read in the previous step 212. The step of "execute message(s)" operation 215 from the appropriate port(s) occurs next. After all of the incoming message(s) have been executed, PAUSE will end in an "end pause" operation 216. At this point, the worker processor 12 will become inactive in an "asleep/inactive" operation 217, and wait until another message header arrives to activate or wake the worker processor 12.

[0083] In summary, all worker processors 12 sleep and PAUSE. The worker processor 12 sleeps, wakes up and reads the I/O register 47 where all neighbor processor 12 write requests are checked, begins PAUSE, and then executes the incoming message. At the end of an incoming message, there is a return (;), or a JUMP to a port 38 with a return (;). The worker processor 12 then goes back to the PAUSE routine, checks the next bit in the I/O register 47 for another message, executes the message, if any, then returns to the worker loop and goes to sleep waiting for more messages.

[0084] Messages are treated as tasks. PAUSE treats incoming messages that are present as waking tasks, and treats ports 38 with nothing on them as sleeping tasks.

[0085] The second occurrence of using PAUSE is shown in Fig. 12, where a processor 12 is "processing a primary function" operation 220. From time to time, the processor 12 will check the I/O register 47 in a "read I/O register" operation 221 to check the incoming status of adjacent processors 12. If the I/O register 47 indicates that there is an incoming message, then the processor 12 will PAUSE in a "call pause" operation 222. The processor 12 will then check the port 38 which was indicated by the I/O register 47 in a "check port for input" operation 223. If there is a message at the designated port 38 as indicated in an "input?" decision operation 224, then the processor 12 will "execute all the code in the incoming message" operation 225, including the four NOPs. After executing the incoming message, there is a "last port done?" decision operation 226 made as to whether there are other ports 38 waiting to send a message. If other port(s) need to be checked for possible input, then oper-

ations are repeated starting at step 223 with a "check port for input" operation. If there is no input there, as indicated by decision 224, the processor 12 will go back to its original task (step 220) - it will never go to sleep under those conditions. After checking all ports 38 in the order of right, down, left, and up (RDLU) for incoming messages and executing any code, the last port will be done (step 226), and the processor 12 will return to its original primary function at step 220.

[0086] Most of the time, communicated sequential processes on different processors 12 is performed by a group of processors 12. If an adjacent processor 12 is not ready to accept a message, then the transmitting processor 12 will go to sleep, or work on another task and will need to poll the I/O register 47 to look for messages. However, most of the time when a group is doing communicated sequential processes on different processors 12, a processor 12 just writes to a port 38 and the adjacent processor 12 just reads from its port 38. A read from all four ports 38 can be performed by a processor 12, then the processor 12 goes to sleep until any one of the four neighbor processors 12 writes; the reading processor 12 needs to check the I/O register 47 to see which processor 12 wrote after being awakened. That processor 12 wakes up with a data read. The processor 12 can read with the A register 40a (data) or the B register 40b (data), or the P register 40c (data or code); reading a whole message with the P register 40c means it will execute an entire message including the four NOPs.

[0087] A processor 12 can read all four ports 38 as a worker, then go to sleep if no message is waiting (Fig. 11). If a processor 12 is busy doing a task (loop), then a PAUSE call can be worked into a task ring to suspend the first task. The main task will be suspended and it will read the four ports 38 which adds four more tasks to the task ring, then it goes back to the main suspended task (Fig. 12).

[0088] Processors with I/O pins connected have bits in the I/O port 39 that set and show the status of those pins. Some pins are read-only and can be read by reading a bit in an I/O port 39. Some pins are read/write and can be read or written by reading or writing bits in the I/O register 47. After a read or write has been completed, the handshake signals are gone and are not readable in the I/O register 47. Pins connected to the wake-up pin handshake circuit of an unconnected port will not be visible in the I/O register 47. If an address that includes a wake-up pin is read, then the processor 12 will wake up when a signal appears on the pin, but if the I/O register 47 is read, then the processor 12 won't see the wake-up handshake. A pin wake-up is connected only to the wake-up circuit, not to the I/O register 47. Therefore, a pin has to be read directly to know if the pin awakened the processor 12. If the pin reads as 0 (zero), then one of the other ports 38 awakened the processor 12. This is how the ROM code in the serial worker processor functions.

[0089] Serial processors have their serial input pin connected to an I/O register bit (bit 17) where they can be read as data, but they are also connected to the handshake line of an unconnected com port. A processor 12 that reads that unconnected com port will wake up when data on the pin tells the processor 12 that a write to the pin or phantom port is taking place. When a real port 38 is read, the processor 12 will sleep until it gets the write handshake signal and will then wake up reading the data. The ROM code distinguishes between a processor 12 being awakened by a pin and being awakened by a port 38 by reading the pin after the processor 12 is awake. If the pin is low, the processor 12 reads the I/O register 47 and pauses. If the pin is high, the processor 12 performs serial boot code.

[0090] When a read or write is started, a read or write handshake bit is raised. When the paired read or write handshake bit is raised by the other processor 12, all bits raised by that processor 12 are lowered, the processors 12 wake up, and the data is transferred. After the data is transferred, any read/write handshake flag bits that were raised are lowered.

[0091] If a processor 12 JUMPs to RDLU (the name of the address of four neighbor ports 38), it tries to read and raises read flags on all four ports 38. It then goes to sleep in essentially a no power mode. When a neighbor (s) writes to one or more of these ports 38 and raises the write flag(s), all port 38 flags are reset and the first processor 12 wakes up. The same thing happens if the processor 12 reads the address using the A register 40a or the B register 40b and a data FETCH or STORE. It goes to sleep until one of the four (or three or two) neighbors wakes it up with data.

[0092] If a wake-up pin is read as part of an address four port 38 read and the pin is not high, then the processor 12 knows that the wake-up came from one of the other three neighbors. Serial boots in ROM on serial boot processors 12 come from a pin which wakes up the processor 12; if the pin is high, then code is loaded from the serial pin and booted by the processor 12.

[0093] There is a path of recommended routes that won't conflict with anything by using X0 packets. X0 stands for execute on the 0 node of the RAM server; messages headed for node 0 on certain paths are defined in ROM. One must not route messages against each other. PAUSE allows messages to be routed safely. A RAM server buffer (on node 6, a node next to the RAM server) allows incoming messages to node 0 of the RAM server to be buffered in RAM on node 6 so that they do not back up and block messages sent out from the RAM server. This is the recommended routing path that doesn't conflict with X0.

[0094] It is important to realize that what is being described here is "cooperative multitasking" between several processors 12. A set of tasks resides on a port 38 or ports 38 and local memory. FETCH B and PAUSE will sequentially examine all ports 38 for incoming executable code and treat ports 38 as tasks. Instructions on a port 38 can be executed directly from the port 38 without loading into the processor's RAM first.

[0095] Although the PAUSE routine between multiple processors 12 has been disclosed herein with reference to Forth, all of the concepts of the PAUSE routine between multiple processors 12 could be applied to other programming languages as well.

[0096] All of the above examples are only some of the examples of available embodiments of the present invention. Those skilled in the art will readily observe that numerous other modifications and alterations may be made without departing from the spirit and scope of the invention. Accordingly, the disclosure herein is not intended as limiting and the appended claims are to be interpreted as encompassing the entire scope of the invention.

Claims

1. A method for communicating between a plurality of computer processors, comprising:

   providing a first processor;
   providing a second processor; and
   sending an input from said first processor to said second processor, wherein said sending does not cause an interrupt in processing functions of said second processor.

2. The method of claim 1, wherein:

   said input comprises a write function from said first processor to said second processor.

3. The method of claim 1, wherein:

   each of said plurality of processors comprises a plurality of communication ports.

4. The method of claim 3, wherein:

   each of said plurality of processors further comprises an input/output (I/O) register.

5. The method of claim 4, wherein:

   each of said input/output (I/O) registers comprises read and write status bits.

6. The method of claim 5, further comprising:

   a step of checking the status of said read and write status bits.

7. The method of claim 1, wherein:

   said plurality of processors comprises an array of processors provided on a die.

8. The method of claim 7, wherein:

   said sending is provided via a data bus between said first processor and said second processor.

9. The method of claim 7, wherein:

   said array of processors comprises at least one interior processor with four adjacent neighboring processors.

10. The method of claim 7, wherein:

    said array of processors comprises at least one processor situated on the perimeter of said array, and wherein said at least one processor further comprises a connection to an input/output (I/O) pin and further comprises an input/output (I/O) status bit.

11. A method of sharing processing tasks between a plurality of processors, comprising:

    providing a first processor;
    providing a second processor;
    providing a communication port between said first processor and said second processor;
    sending an input from said first processor to said second processor; and
    receiving said input by said second processor from said first processor, wherein said sending does not interrupt the processing functions of said second processor.

12. The method of claim 11, wherein:

    said sending occurs when said second processor is executing a task.

13. The method of claim 12, wherein:

    said receiving is completed when said second processor temporarily pauses said executing and accepts said sending from said first processor.

14. The method of claim 12, wherein:

    said sending an input from said first processor further comprises setting an input flag bit high.

15. The method of claim 12, wherein:

    responding to said received input by said second processor comprises executing code of said received input directly from said port.

16. The method of claim 15, wherein:

said executing code directly from said port is performed in the absence of storing said code to a memory location prior to said executing code directly.

17. The method of claim 12, further comprising:

a software program, wherein said software program comprises a step of temporarily pausing said executing a task of said second processor, and checking said port for potential input from said first processor.

18. A method, comprising:

sending an input from a first processor to a second processor, wherein said second processor is inactive at time of said sending;
awakening said second processor to receive said input;
checking an input/output (I/O) register by said second processor to determine the source of said input;
receiving said input by said second processor from said first processor; and
responding to said input from said first processor by said second processor.

19. The method of claim 18, wherein:

said method is executed by a software loop.

20. The method of claim 18, wherein:

said awakening comprises a message header which is safe to throw away after said awakening.

21. The method of claim 18, wherein:

said checking comprises determining the status of read and write handshake status bits of adjacent processors.

22. The method of claim 21, wherein:

said receiving is followed by lowering the read and write handshake status bits of said first processor and said second processor.

23. The method of claim 18, wherein:

said acting is followed by said second processor returning to an inactive mode.

24. The method of claim 18, wherein:

said awakening is caused by a multiple port read function.

25. The method of claim 18, wherein:

said awakening is caused by a pin.

26. The method of claim 18, wherein:

said receiving comprises reading said input as a data statement.

27. The method of claim 18, wherein:

said method is located in ROM.

28. The method of claim 27, wherein:

said method is part of a boot up task in ROM.

29. A computer readable medium having code embodied therein for causing an electronic device to perform the steps of claim 18.

30. A method for communicating between a plurality of computer processors, comprising:

providing a first processor;
providing a second processor, wherein said second processor is in an alert but inactive status;
providing an I/O register for each of said plurality of computer processors;
sending an input from said first processor to said second processor, wherein said sending causes said second processor to change to an active status;
reading the I/O register of said second processor to determine from which processor said input was sent; and
directly executing said input by said second processor.

31. The method of claim 30, further comprising:

reading said I/O register of said second processor an additional number of times to determine if additional inputs have been sent to said second processor; and
executing said additional inputs by said second processor.

32. The method of claim 31, wherein:

said additional inputs were sent from a third processor.

33. A processing system, comprising:

an array of interconnected computer proces-

sors, wherein each processor further comprises:

an I/O register;
a communication port located on each of four sides of said processors;
a sending mechanism for sending an input to other said processors; and
a receiving mechanism for receiving an input from other said processors;

a monitoring mechanism in which each of said processors receiving said input can determine the source of said input; and
an executing mechanism in which each of said processors receiving said input can respond to said input.

34. The system of claim 33, wherein:

said sending mechanism comprises a first port on a first processor sending the input, said first port being located adjacent to an intended receiving processor; and
said receiving mechanism comprises a second port on said intended receiving processor, said second port being located adjacent to said first processor.

35. The system of claim 34, wherein:

said receiving mechanism further comprises a latching mechanism to receive said input directly from said first port to said second port.

36. The system of claim 33, wherein:

said monitoring mechanism of a first processor can suspend an active executing task of said first processor in order to determine if an input is attempting to be sent by a second processor.

37. The system of claim 33, wherein:

said input is received by a receiving processor directly from a port of a sending processor.

38. The system of claim 33, wherein:

said sending mechanism has the ability to send said input from one processor to a non-adjacent receiving processor.

Fig. 1

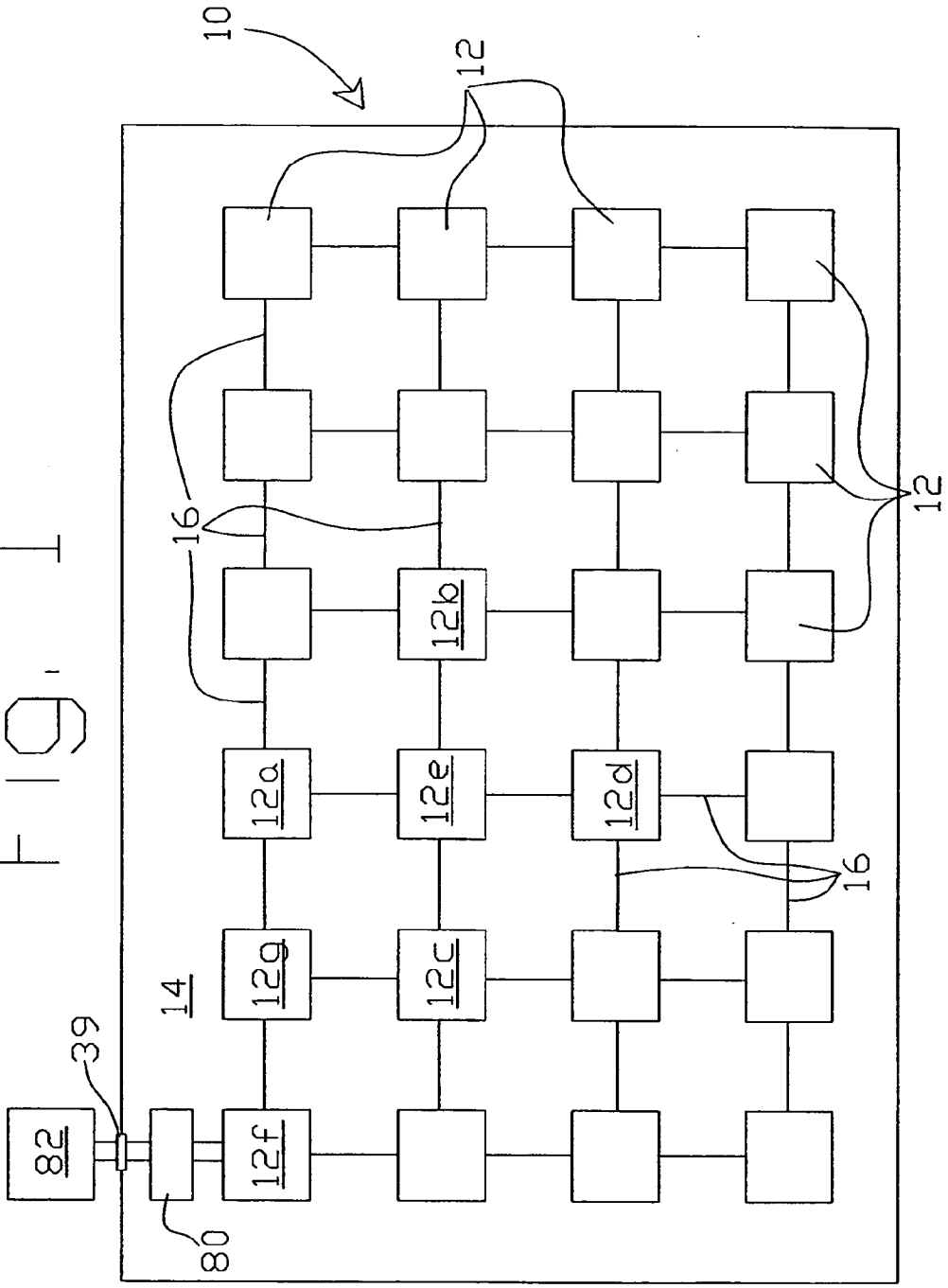


Fig. 2

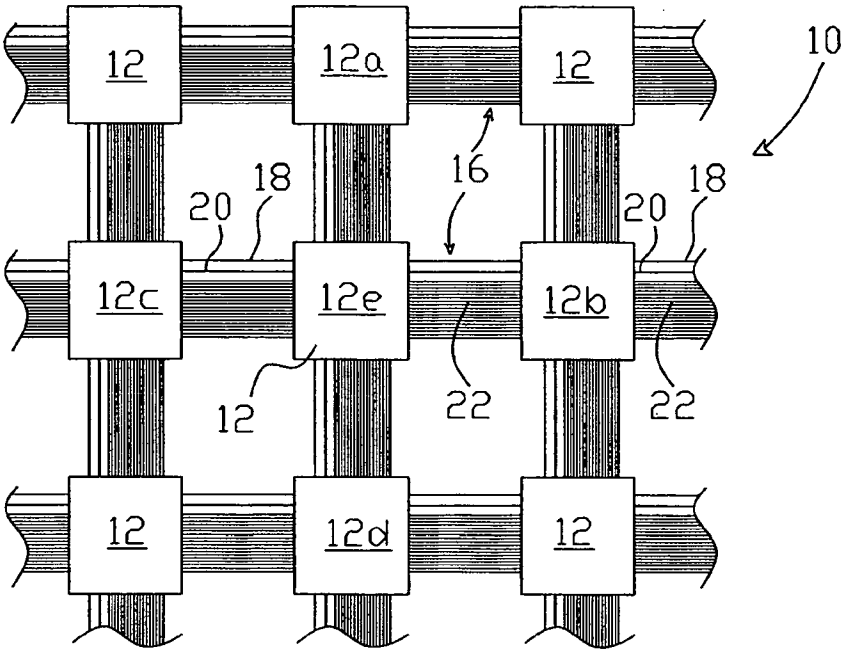

12
12a
12
10
16
18
20
18
20
12c
12e
12b
12
22
22
12
12d
12

Fig. 5

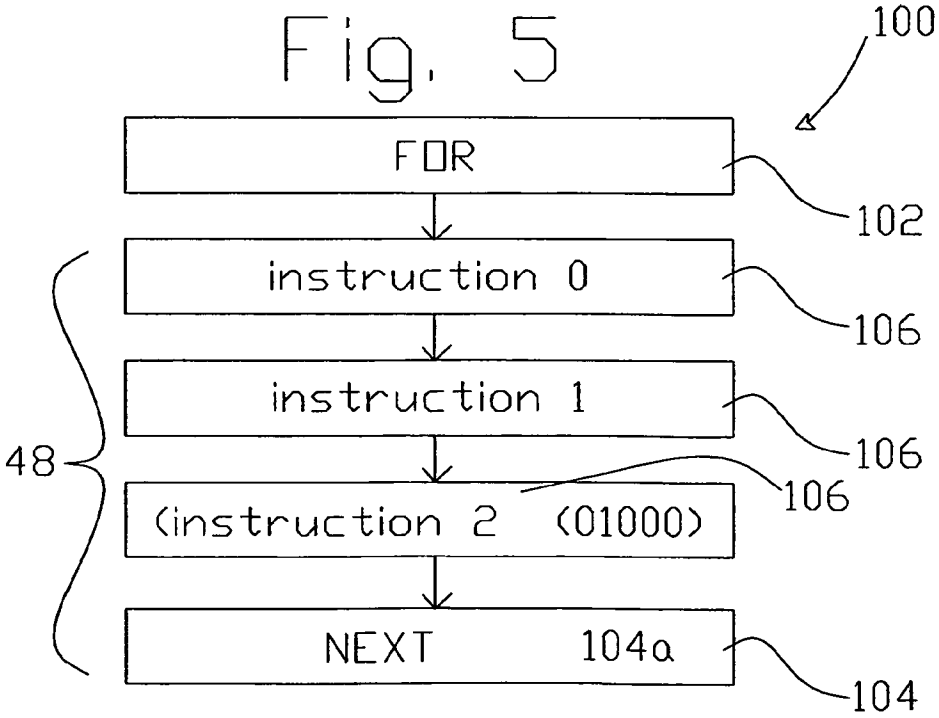

100
FOR
102
instruction 0
106
instruction 1
106
48
106
(instruction 2 (01000)
NEXT 104a
104

Fig. 3

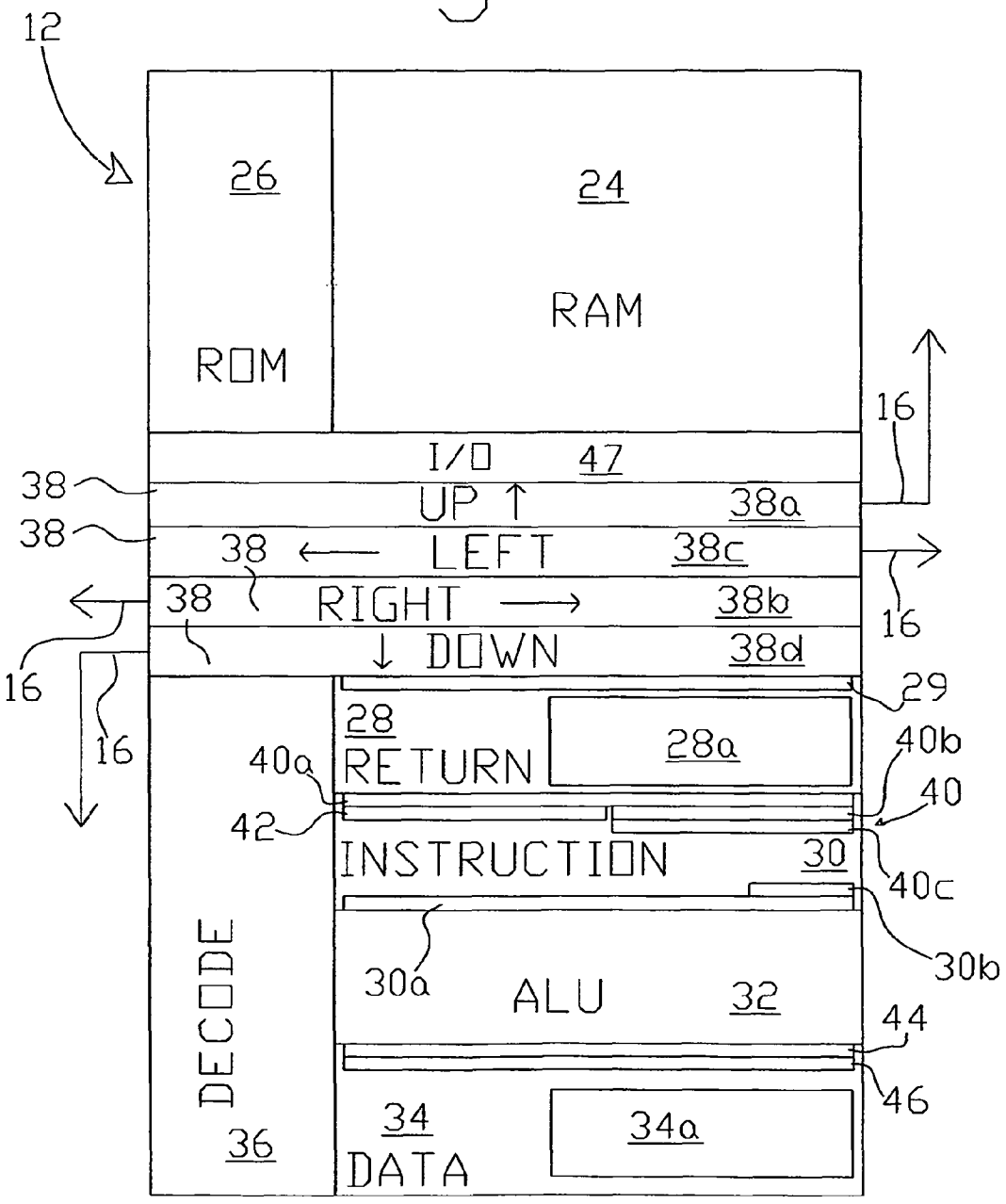

12
26
ROM
24
RAM
16
I/O 47
38
UP ↑ 38a
38
38 ← LEFT 38c
38 RIGHT → 38b
16
38
↓ DOWN 38d
16
16
29
28
RETURN
28a
40a
40b
42
40
INSTRUCTION 30
40c
30b
30a
ALU 32
44
46
DECODE
36
34
DATA
34a

Fig. 4

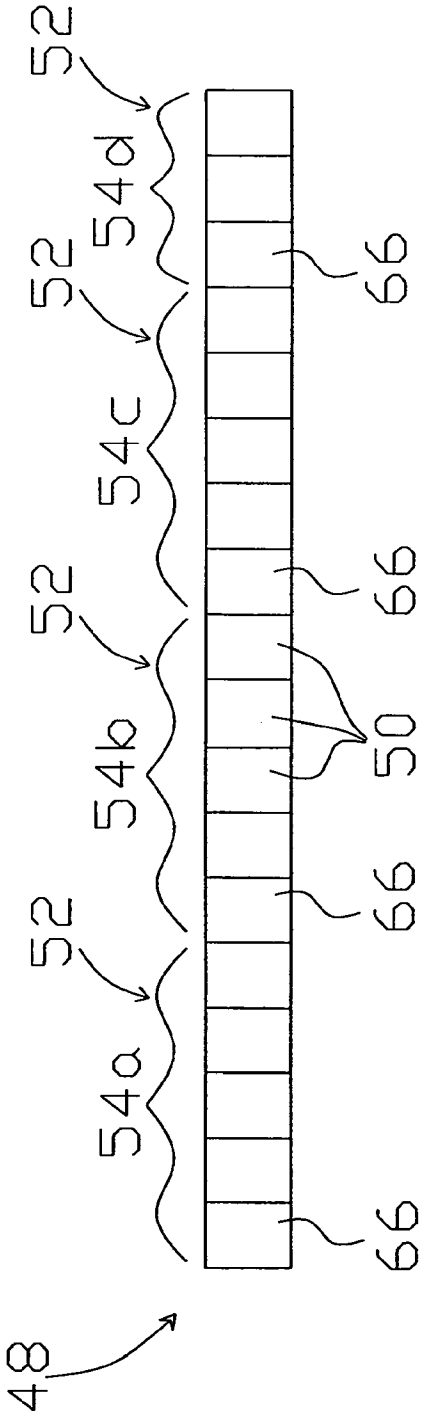


Fig. 6

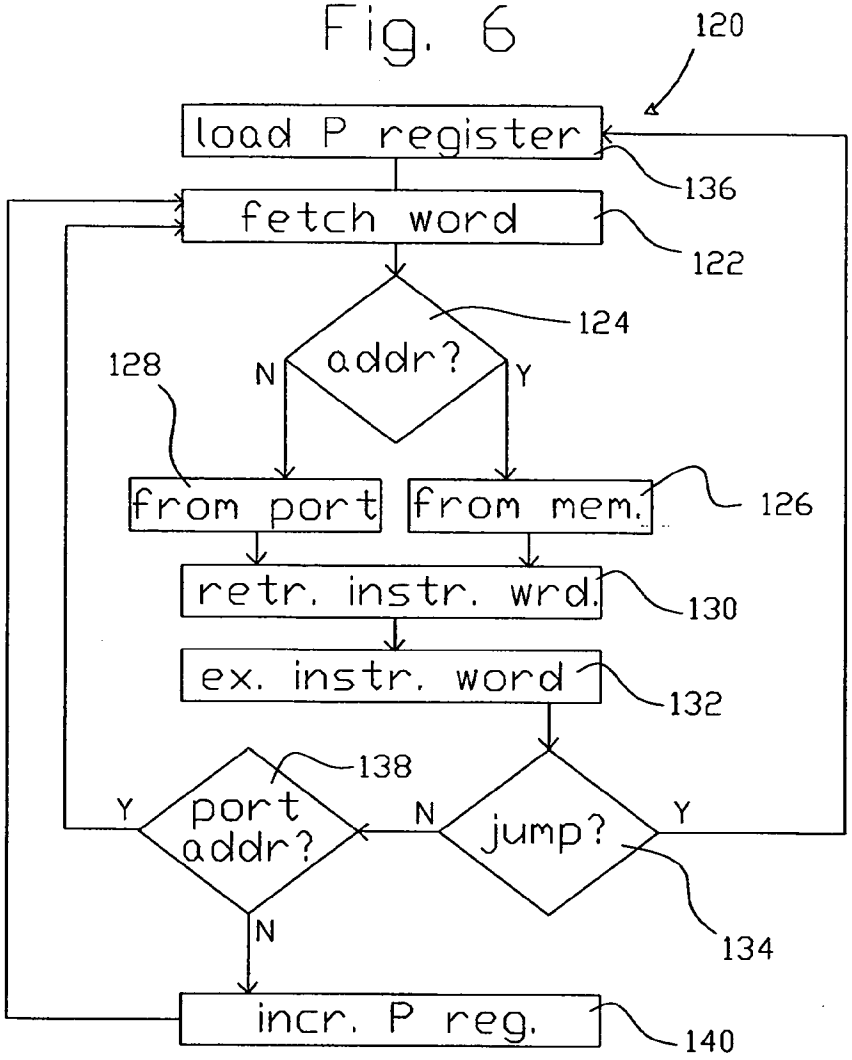

120
load P register
136
fetch word
122
124
addr?
N
Y
128
from port
from mem.
126
retr. instr. wrd.
130
ex. instr. word
132
138
port addr?
Y
N
jump?
Y
134
N
incr. P reg.
140

Fig. 7

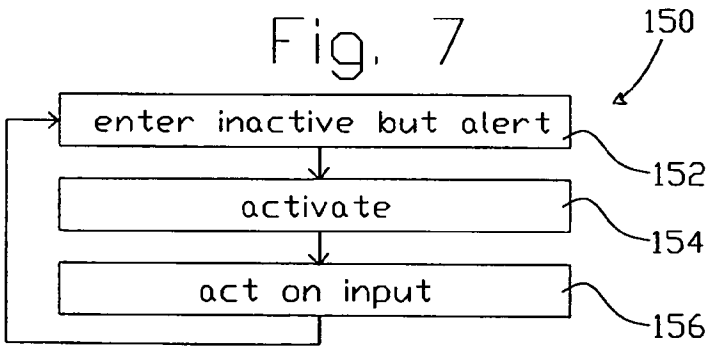

150
enter inactive but alert
152
activate
154
act on input
156

Fig. 8

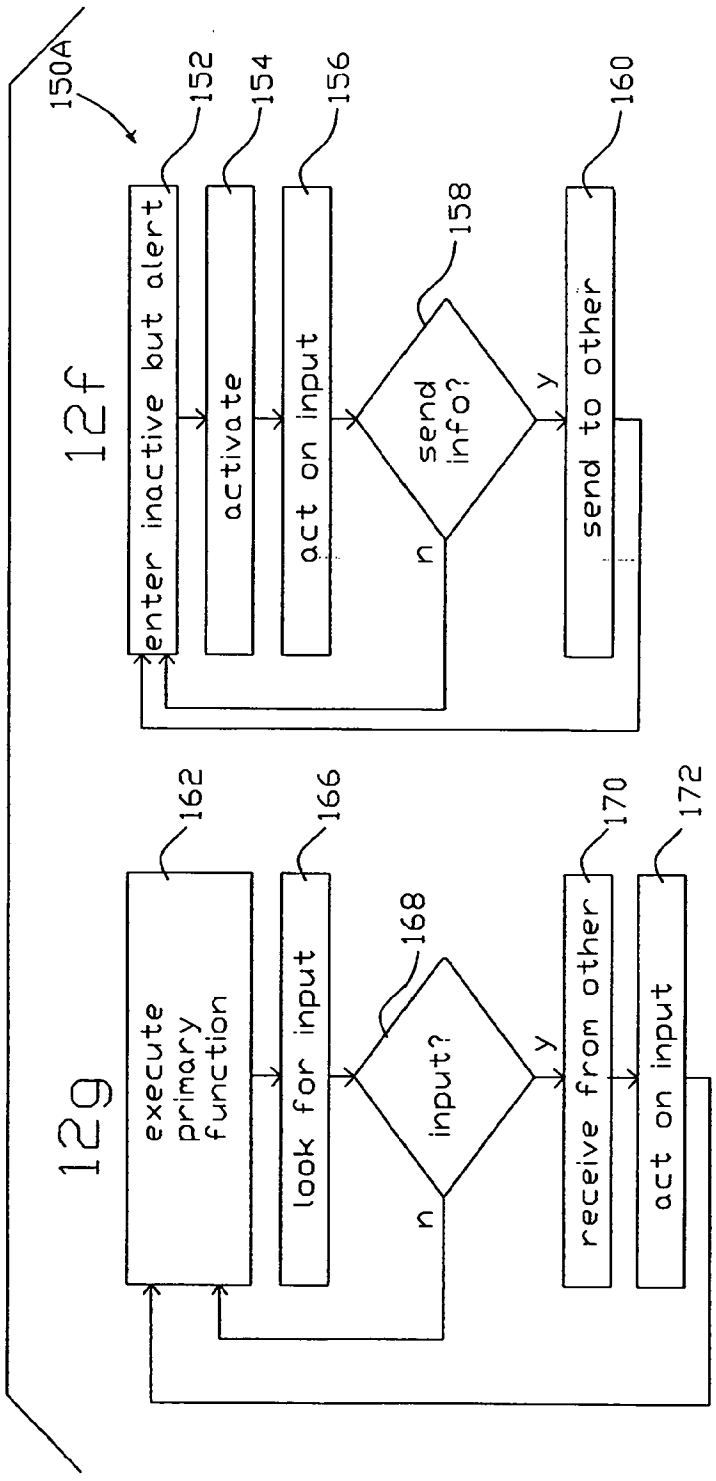

150A
12f
enter inactive but alert
152
activate
154
act on input
156
send info?
158
y
n
send to other
160
12g
execute primary function
162
look for input
166
input?
168
y
n
receive from other
170
act on input
172

Fig. 9

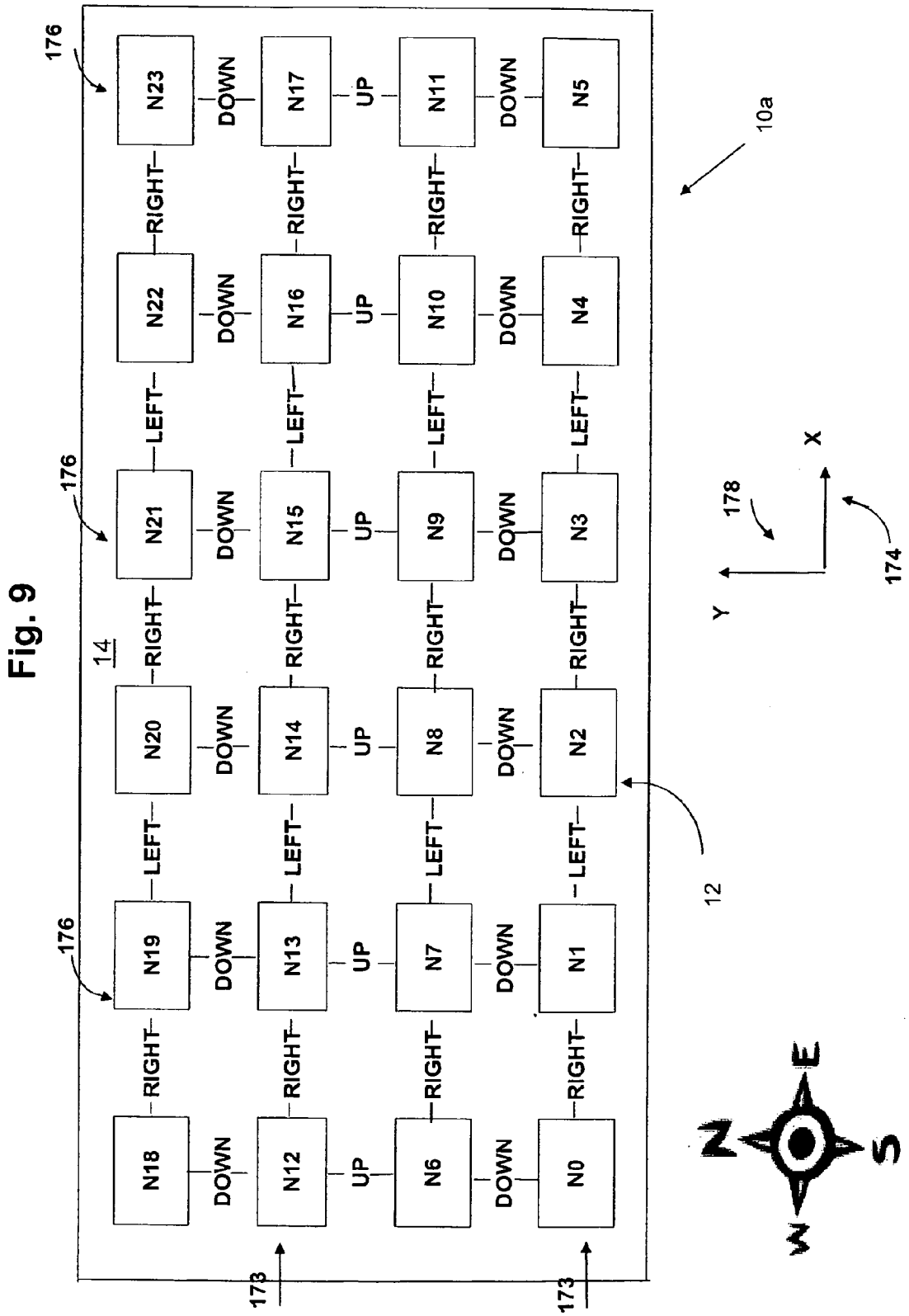
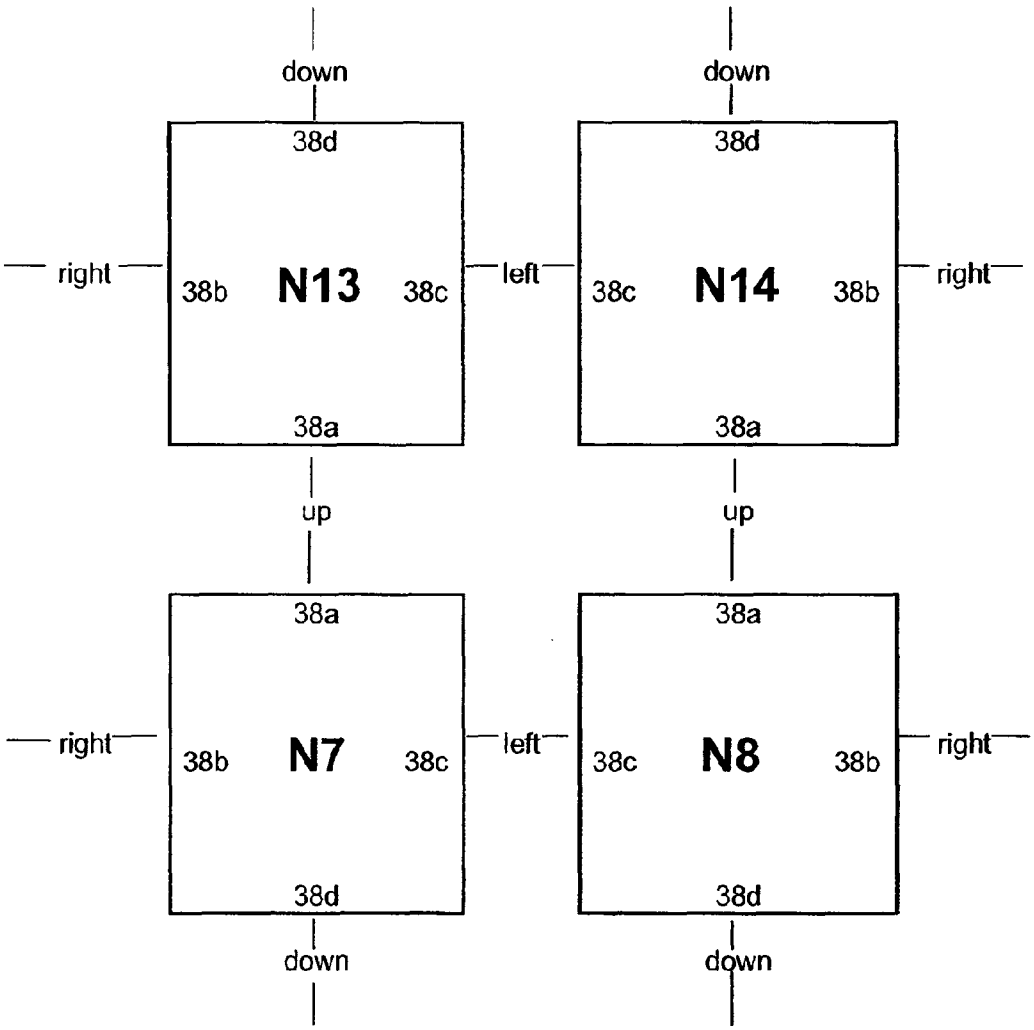
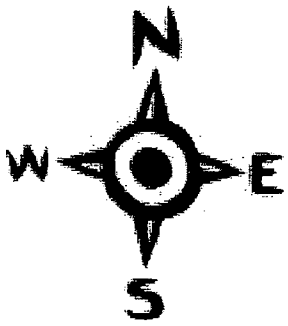

176
N23
DOWN
N17
UP
N11
DOWN
N5
RIGHT
RIGHT
RIGHT
RIGHT
N22
DOWN
N16
UP
N10
DOWN
N4
LEFT
LEFT
LEFT
LEFT
176
N21
DOWN
N15
UP
N9
DOWN
N3
RIGHT
RIGHT
RIGHT
RIGHT
14
N20
DOWN
N14
UP
N8
DOWN
N2
LEFT
LEFT
LEFT
LEFT
176
N19
DOWN
N13
UP
N7
DOWN
N1
RIGHT
RIGHT
RIGHT
RIGHT
N18
DOWN
N12
UP
N6
DOWN
N0
173
173
12
10a
178
X
Y
174
E
N
S
W

Fig. 9a


down
down
38d
38d
right
38b
N13
38c
left
38c
N14
38b
right
38a
38a
up
up
38a
38a
right
38b
N7
38c
left
38c
N8
38b
right
38d
38d
down
down


N
W
E
S

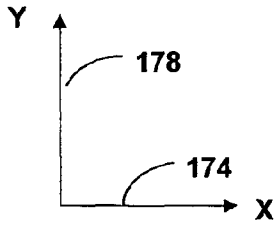

Y
178
174
X

Fig. 10

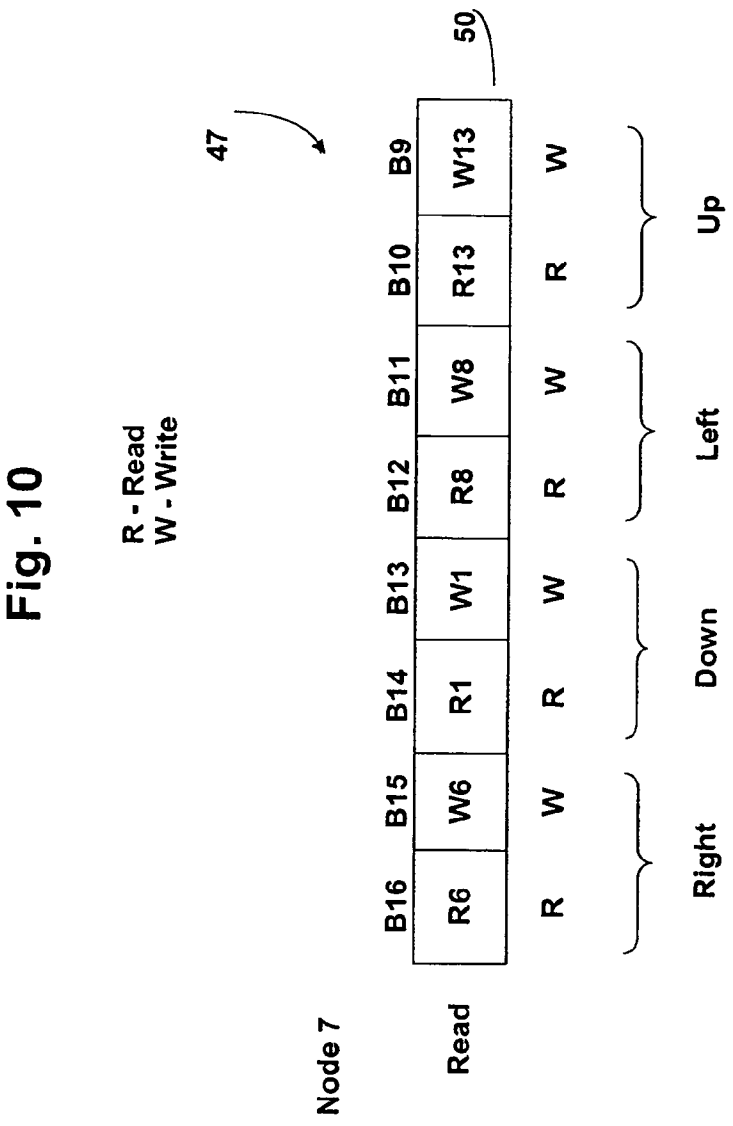

R - Read
W - Write
47
50
Node 7
Read
B16
B15
B14
B13
B12
B11
B10
B9
R6
W6
R1
W1
R8
W8
R13
W13
R
W
R
W
R
W
R
W
Right
Down
Left
Up
RDLU

Fig. 11

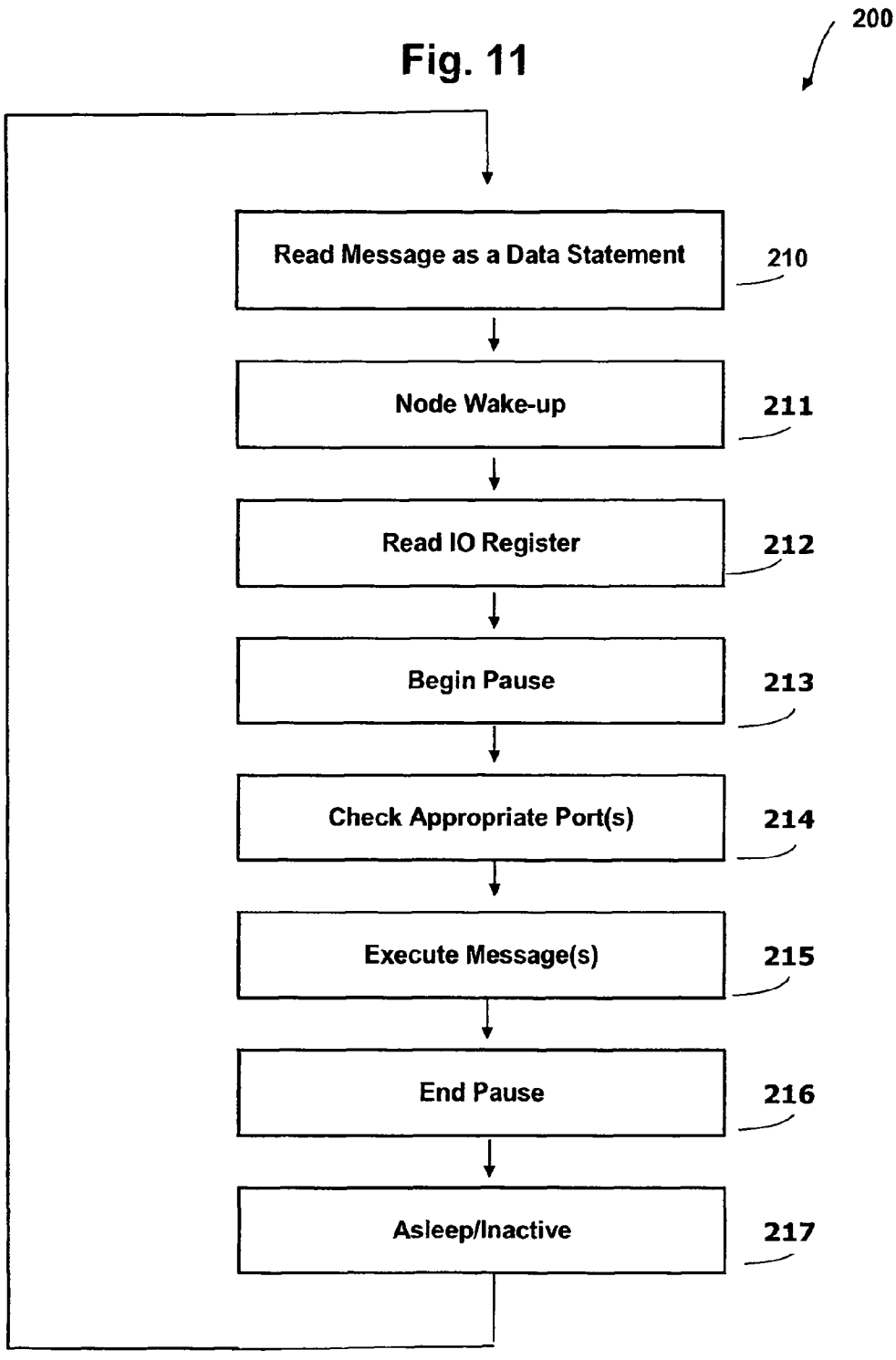
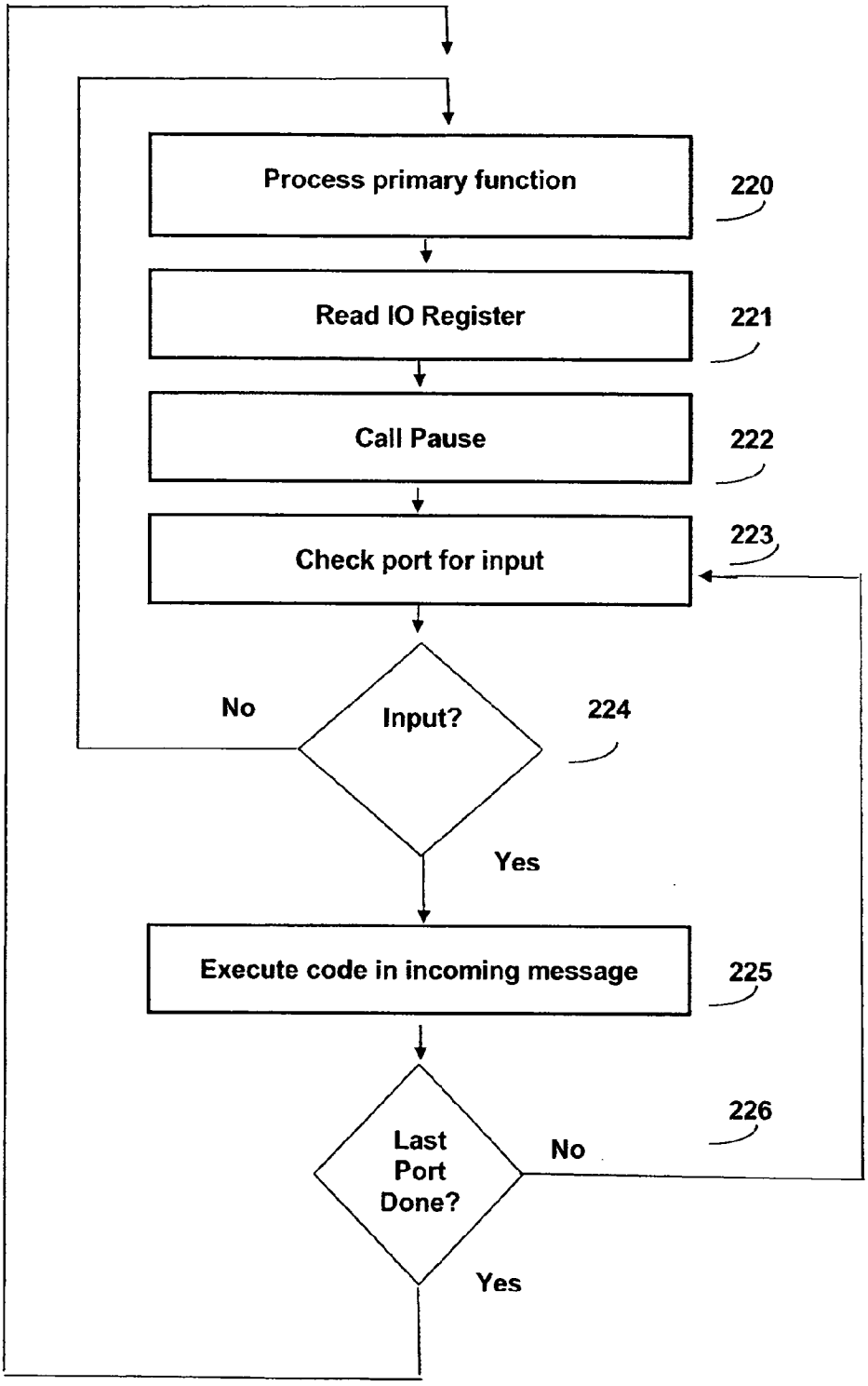

200
Read Message as a Data Statement
210
Node Wake-up
211
Read IO Register
212
Begin Pause
213
Check Appropriate Port(s)
214
Execute Message(s)
215
End Pause
216
Asleep/Inactive
217

Fig. 12


Process primary function
220
Read IO Register
221
Call Pause
222
Check port for input
223
Input?
224
No
Yes
Execute code in incoming message
225
Last Port Done?
226
No
Yes

REFERENCES CITED IN THE DESCRIPTION

This list of references cited by the applicant is for the reader's convenience only. It does not form part of the European patent document. Even though great care has been taken in compiling the references, errors or omissions cannot be excluded and the EPO disclaims all liability in this regard.

Patent documents cited in the description

- WO 60849498 A **[0001]**
- WO 60818084 A **[0001]**
- WO 11441818 A **[0001]**
- WO 60797345 A **[0001]**
- WO 60788265 A **[0001]**
- WO 11355513 A **[0001]**

Non-patent literature cited in the description

- **FARZAN FALLAH et al.** Standby and Active Leakage Current Control and Minimization in CMOS VLSI Circuits. 2004 **[0013]**